トランジスタ技術
SPECIAL
エレクトロニクスの基礎と実用技術を凝縮したフィールド・ワーク・マガジン

No.79

特集

ハード設計の原点を見直して，できる回路設計者になろう

初歩のHDL設計学習帳

トランジスタ技術

SPECIAL

特集

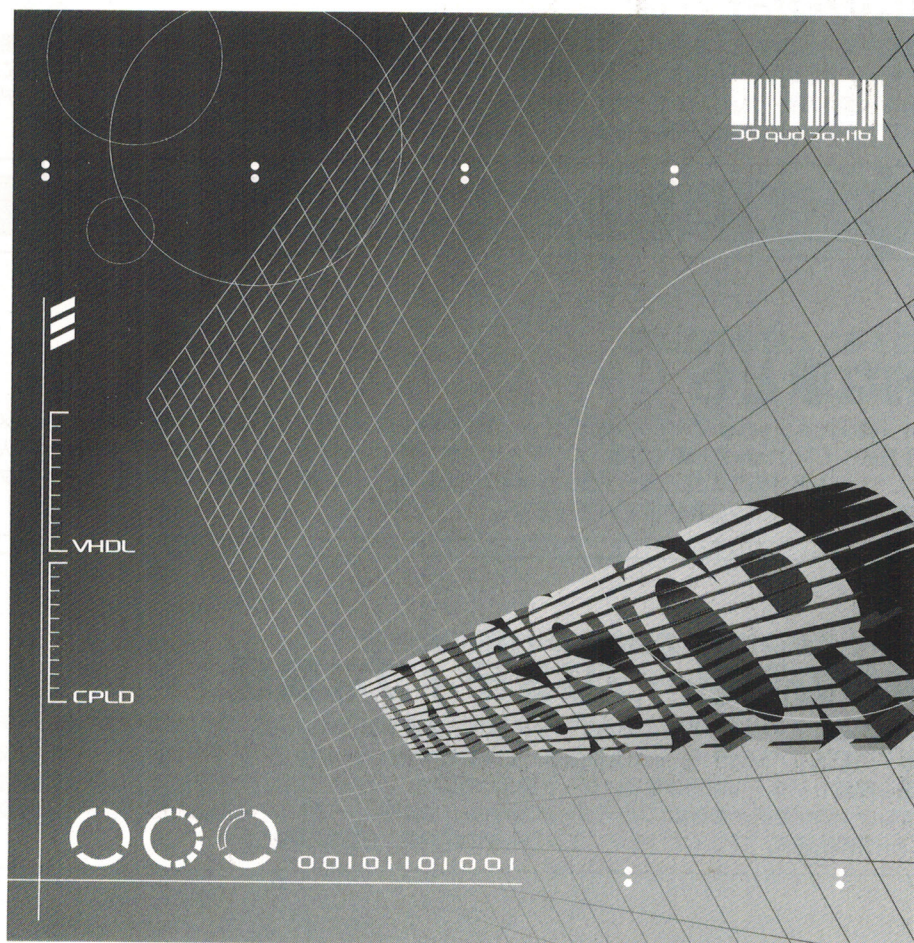
ハード設計の原点を見直して，できる回路設計者になろう

初歩のHDL設計学習帳

HDL(ハードウェア記述言語)という言葉も最近ではかなり一般的なものになってきました．HDLは主としてディジタル回路をテキストで表現するために考えられた言語です．アナログ回路の設計やプリント基板上の回路の設計にHDLが使われることも皆無ではありませんが，まだこれらの例は少数です．FPGAやCPLDなどのフィールド・プログラマブルなIC，半導体メーカーが製造する汎用IC，スタンダードセルやゲートアレイなどのカスタムICを設計するときの必需品であるHDLをマスタして，次世代でも生き残れるエンジニアになりましょう．

CONTENTS

特集 ハード設計の原点を見直して、できる回路設計者になろう 初歩のHDL設計学習帳

表紙デザイン／アイドマ・スタジオ　表紙イラスト／柴田 幸男　本文イラスト／神崎 真理子

第1章　VHDLの来し方行く末

ハードウェア記述言語を理解する

吉澤　清

HDLとは

HDL(ハードウェア記述言語)という言葉も最近ではかなり一般的なものとなってきました.「**ハードウェア記述言語**」は,主としてディジタル回路をテキストで表現するために策定された「言語」です.

「言語」というと,だれもがすぐに思い浮かべるのは「BASIC」や「C」などのコンピュータ用「プログラミング言語」ではないでしょうか.

〈図1-1〉プログラムとVHDLコードの違い

	プログラム	VHDLコード
目的	コンピュータ上の 処理手順の記述	ディジタル回路の 構造/機能記述
テキストのもつ意味合い	フローチャートを テキスト記述したもの	回路図を テキスト記述したもの
動作の仕方	**シーケンシャル動作** 1行ずつ順次実行される	**コンカレント動作** 記述された機能すべてが 同時並行的に動作する
その他	**コンピュータへの依存が不可欠** その反面,豊富なリソースが 使用可能	**自己完結が可能** その反面,必要なものは すべて用意する必要がある

ハードウェア記述言語

Hardware Description Language (HDL).ハードウェア(電子回路)の構造/機能をテキストにより記述することを目的とした言語体系.現在の主流はVHDLやVerilog－HDL.最近ではSpecCなどのC言語から派生したシステム記述言語と呼ばれるものも登場している.

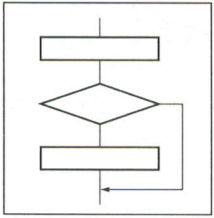
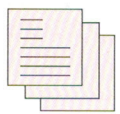
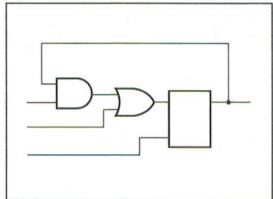
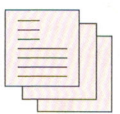
シーケンシャル動作

いろいろな処理/動作が順を追って一つずつ行われるさまを示す.

コンカレント動作

いろいろな処理/動作が同時に並行して行われるさまを示す.実際の電子回路の動作は同時並行的である.シーケンシャル動作の対語.

コンピュータへの依存が不可欠

プログラムは,コンピュータ上で走ることを想定して作られている.したがってプログラムはコンピュータなしには働くことができない.その代わり,プログラムはコンピュータがもっているすべての資源(インターフェース,メモリなど)を使うことができる.

自己完結が可能

VHDLにより設計されたチップは,それ自身が完成された存在であり,独立して動作をすることが可能である.しかし反面,必要な資源については,他を頼ることはできず,すべて自前で用意しなければならない.

しかし,「ハードウェア記述言語」は「プログラミング言語」とは異なります.「プログラミング言語」はコンピュータの上で走るプログラムを作成するために使われます.いっぽう,「ハードウェア記述言語」はディジタル回路,つまりコンピュータを含むさまざまな電子装置の回路を設計するために用いられます(**図1-1**).

このため,**HDLコード**を書こうとする場合には,かならずディジタル回路に関する知識が不可欠となります.なぜならば,設計の途中段階ではシミュレータ上での検証も可能ではあるものの,HDLで設計を行う場合に,最終的にできあがるのは電子回路(主としてIC)だからです.

HDLコード
ハードウェア記述言語で書かれた回路を記述したテキストのこと.

一時,「HDLさえマスタすれば回路設計はできる」という誤った考え方が流布した時代がありましたが,それは間違いです.正確には「回路図の描き方を憶える」かわりに「HDLの書き方を憶える」ことが必要になったというところでしょうか.

というわけで,「デジタル回路」の教科書や「電子設計」関連の書籍が不要になったわけではありません.HDLについて学ぶと同時に,これらの分野に関する造詣も深めていく必要があります.さらには,HDL設計のターゲットの多くがICチップであることを考えると,「半導体」に関する知識もあるに越したことはないと言えるでしょう.

HDLの使途

アナログ回路の設計やプリント基板の回路の設計にHDLが使われることも皆無ではありませんが,まだこれらの例は少数です.

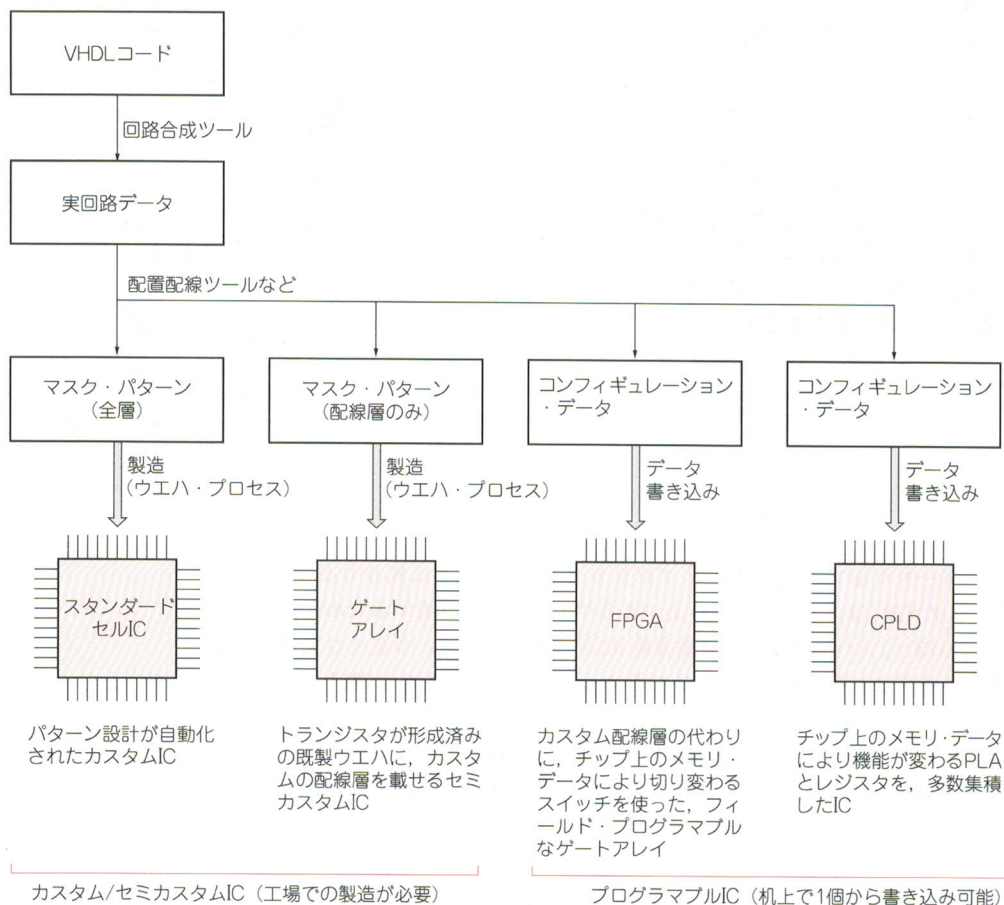
HDLの本領は,やはりディジタルICの設計ということになります(**図1-2**).

具体的には,

- 半導体メーカーが製造する汎用IC(量産品)
- スタンダード・セルやゲートアレイなどの**カスタムIC**
- FPGAやCPLDなどのフィールド・プログラマブルなIC

などがHDL設計のターゲットになります.

カスタムIC
半導体メーカーが不特定多数の顧客に対して販売する汎用ICに対して,特定の顧客の仕様に基づいてオーダメードで作られるICをカスタムICと呼ぶ.アナログ系や特殊な回路に関しては専用にパターンが設計されることもあるが,一般的なディジタル回路はスタンダード・セル方式で実現されることがほとんどである.

〈図1-2〉 VHDLコードの利用(ディジタルICの設計)

HDL誕生の背景,そして現在

i4004
1971年に米インテル社が発表した,世界初の1チップ・マイクロCPU.製造プロセスはP-MOSであり,数百kHzのオーダーの2相クロックで動作するものであった.もちろんまだ単一電源動作ではない.日本のビジコン社の発注により開発されたことは有名.パッケージのピン数に制限があったためマルチプレックス・バスが採用されたなどのエピソードは「我が青春の4004」に詳しい.1974年のトランジスタ技術誌にi4004を使ったマイクロコンピュータ製作の特集記事があった.

HDLが誕生した最大の理由は,半導体製造技術の爆発的ともいえる急速な進歩でした.電子業界ではよく耳にする「このチップは設計ルールが0.3μ(ミクロン)」とかいうあれです.

IC上の回路は2次元的な広がりをもっているため,加工技術の微細化が進むと,チップ上に詰め込むことができる回路の量はその二乗に比例して膨らんでいきます.また,1個のチップの面積の大型化も集積度の向上に拍車をかけました.

1960年代末に作られた世界初のワンチップ・マイクロコンピュータi4004は千ゲート程度のチップでしたが,21世紀初頭の現在では数百万ゲートのICが製造可能な状況にあります.ここ30年ほどでICのゲート数は3桁以上増加したことになります.

ここで,数百万ゲートのICの回路図というものを考えてみましょう.かりに完全にフラットな設計(階層設計を使わない)をしたとして,1枚の回路図に1000個のゲートを描いたとすると,数千枚の回路が必要となります.数千枚の回路図

の厚みは200ページの本に換算して数十冊分です．これは十センチ単位の厚さということになります．

このような回路図相手では，特定の一カ所を探し出すというだけで難作業となります．

1980年代，その時点までの半導体製造技術のトレンドから近い将来，回路図ベースの大規模集積回路設計が行き詰まることが予測され，その対策が検討されました．その結果として現れたのが「ハードウェア記述言語(HDL)」なのです．

HDLを用いることにより，回路図設計の場合と比較して数倍～数十倍の回路規模に対応することが可能になりました．これだけでもたいへんなことです．

しかし，数百万ゲートという回路規模はすさまじいもので，HDLのみで対処可能な代物ではありませんでした．そして近年ではIPの利用，とくに既存のCPUやDSPの利用が設計効率向上のために行われているようです．それでもSOC(システム・オン・チップ)の開発にはコストがかかりすぎて実際的ではないなどという話もあり，今後も設計手法の革新は続いていくのでしょう．

とはいえ，回路設計者のなかで，ここまでの大規模設計に携わる方はごく少数でしょう．そういった意味では数千～数万ゲート規模のディジタル回路の設計を格段に容易にしてくれるHDLは，一般の回路設計者の大多数に福音をもたらす「言語」ということができるでしょう．

SOC

System On Chipのこと．1万ゲート以下の規模のチップは，感覚的にまだ部品というイメージがあった．しかし，現在ではすでに数百万ゲートのチップが実用化されており，これを使えば大規模なシステムさえも1チップに収めることが可能である．ただし，そのシステムの設計は回路規模が巨大であることから至難であり，チップのゲート規模の拡大が続く限り設計者(チーム)の苦労は続く．

HDL設計のメリット，Verilog-HDLとVHDL

HDL設計のメリットを**図1-3**に示します．

現時点において実用段階にあるHDLとしては，Verilog-HDLとVHDLの2種類があげられます．Verilog-HDLは米国のケイデンス社で開発された言語であり，

〈図1-3〉VHDLのメリット(回路図設計やほかの言語とくらべて)

- 真のコンカレント言語である
  - コード内の記述が並行して同時に働くことを明快に記述できる言語
    - → 実回路の動作を忠実に表現することが可能
- 移植性が高い
  - あらゆるテクノロジ（IC製造技術）への展開が可能
    - → デバイス・マイグレーション（メーカ間の移行を含む）が容易
    - → 設計資産の再利用が容易
- 参照に特殊なツールは不要
  - VHDLコードは一般的なテキスト・ファイルの形で取り扱われる
    - → 表示のスクロールも高速（図画描画には時間がかかる）
- 検索が容易
  - モジュール名や信号名による位置検索が容易…テキスト・エディタの検索機能で可能
- 設計効率が向上する
  - …使いはじめれば手離せないツール
  - 記述効率が高い
    - 効率向上のための言語仕様（信号のベクタ表現/処理の繰り返し表現/シーケンシャルな記述/etc.）
      - → 回路図設計の数倍～数十倍
  - エントリ効率が高い
    - コンピュータ上で図を描くのではなくテキストを打ち込めばよい
  - コード・ジェネレータが作りやすい
    - ファイル形式はテキスト・ファイル/言語仕様が明確
      - → ストリング処理ができるプログラム言語があればコードの自動生成が可能

≪錦の御旗≫

VHDLは米国防総省のVHSICプロジェクトから生まれた言語です．いずれも米国生まれです．

HDLの黎明期，シミュレータの開発において先行したVerilog-HDLが市場を席巻しました．文字通りHDLイコールVerilog-HDLという世の中だったのです．

しかし，同時に巷にはつぎのような噂が流れていました．「間もなくVHDLの開発ツールがリリースされる．VHDLのツールが登場すれば，市場はVHDLへ移行する」と．

VHDLの開発ツールがリリースされると，噂に違わずVHDLは爆発的に広まりました．回路設計の分野において，すでにある言語が市場形成を果たした後に新しい言語が市場へ参入したという状況を考えると，これは異常ともいえることでした．これは，VHDLへの期待の高さ，そしてVHDLが実用状態となるまで，HDLの導入を待った企業が多かったことを示していました．

このようにして，Verilog-HDLとVHDLの競争は劇的に幕を開けました．

さて，そのような状況下でVerilog-HDLはどうなったのでしょうか．実はVHDLが市場への参画を果たしたあとも，Verilog-HDLは健闘を続けていたのです．

これには市場へ早期参入した強みや，市場の柔軟性に助けられてという面もあるでしょう．しかし，なによりの要因はVerilog-HDLがHDLとしてVHDLに比肩する秀れた言語であったことにあるものと思われます．このことは，その後もVerilog-HDLとVHDLがほぼ互角の競争を展開していることからも明らかです．時間と市場自身がそのことを証明してくれたわけです．

世の中には「すべての規格は統一されたほうがよい．当然，HDLも一本化されるべきである」との論があります．HDLが統一されていれば，入門しようとするときに迷うこともありませんし，ちょうど欲しかったIPがほかの言語で書かれていたがために使えないといった問題も発生しません．HDLが複数存在することは悪なのでしょうか．

いいえ，そうとばかりは言えないようです．一つには言語間に競争原理が働くと

VHSICプロジェクト

Very High Speed Integrated Circuitプロジェクト．ペンタゴン(米国防総省)の次世代高速IC開発計画のこと．国防総省というと目的が軍事利用に特定されるようにも思えるが，実際にはこのプロジェクトは米国の産業界の活性化に大きく貢献した国家的プロジェクト．現在我々が，VHDLを使うことができるのも，元をただせばペンタゴンのおかげと言えるのか？．

最近では米国は日本に技術を売ってくれないという話もあるが，少なくともHDLに関してはそうではない．米国が大量の資金を投じて生み出したHDLが非常に廉く使うことができて，その効果は少なくないのであるから，これを使わない手はない．

市場の柔軟性

市場というものは，いろいろな考え方をもつ個人の集合体であるから，外部からの刺激に対する反応は単純に予測できるようなものではない．プロデュースする側が市場に対して分析を行いそれに対する戦略を展開したとしても，かならずしもその意図通りになるとは限らない．市場は，気まぐれ気ままな生き物のようなものといえるのではないだろうか．

IP

Intellectual Propertyのこと．直訳では知的財産権となるようであるが，電子業界では既製の汎用の回路設計データ(今日では通常HDLで記述される)のことを指す．このSOC時代において，数百万ゲートの回路を一から設計することは到底不可能なため，IP(オンチップCPU/DSPを含む)を利用して，設計効率(いかに短期間で多くのゲートを消費するか)を向上することが，大規模設計を行う場合の一般解となりつつある(しかし，それでもまだまだ足りないというのが実状か？)．

いうことがあります．HDLの開発環境はここ十数年ですばらしい発展を遂げました．もちろん，半導体の製造技術のトレンドからくるプレッシャや，大規模設計現場からの期待によるところもあったでしょう．しかし，これに加えてVerilog-HDLとVHDLの間の競争がさらなるドライブをかけたこともまた事実です．

競争原理が働くには両者の力量にあまり大きな差があることは望ましくありません．幸いなことは，Verilog-HDLとVHDLは互いに互角に渡り合えるライバルであったことです．互いが互いの発展に貢献することができたのですから．

両者がよい競争関係になければ，**HDLの開発環境**がここまで急速に進歩することもなかったでしょうし，その価格がここまで下がりもしなかったことでしょう．

二つ目の利点として，設計者の選択肢が広がることがあげられます．人それぞれ趣味趣向は異なるため，ある程度の選択の自由があったほうが自分に合った言語をチョイスできるという意味でよいということができるでしょう．

「Verilog-HDLにするかVHDLにするか，それが問題だ」とお悩みの方，確率50％の賭の世界へようこそ．この選択を難しいと感じている方は，この選択が自らの仕事に大きな影響を及ぼす反面，その選択を裏打ちすべき根拠の希薄さを感じるのではないでしょうか．しかし，それでも判断は下さなければなりません．

それぞれの言語の将来性や回路設計現場の未来を確実に予想することが事実上困難であることを考えると，この決断はまさに賭けであると言えるでしょう．しかし，ここで思い出してください．Verilog-HDLやVHDLなどのHDLは**回路設計の道具**に過ぎません．あくまでも目的は回路設計なのです．

大切なのは，言語としてのHDLではなく，HDLを使って回路設計を行う方法を学ぶことです．もし手をつけた言語が衰退するようであったなら，ほかの言語にシフトすればよいのです．言語間の移行も苦痛を伴うことでしょう．けれども，回路設計よりHDL設計への移行や，ゼロからのHDL設計への入門よりたいへんということはないはずです．まあ，とにかく賽を振ってみようではありませんか．そして運を天に任せ，自分のできる最大限の努力をする．この世の中，往々にして決断することよりその後の努力により結果が天と地の開きになってしまうことがあるのですから．

というわけで(？…)，なんとか口実を見つけてVHDL(あるいはVerilog-HDL)を選んでしまいましょう．たとえば，

▶会社で広く使われているVHDLを選択する
　ツールも揃っているし，仕事にもすぐ役に立つ．教えてくれる人もいる．
▶求人の募集要件として掲載されていたVHDLを選択する
　VHDLが使えると条件がよくなる．HDLはビジネスマンの武器である．
▶取引先よりVHDLで設計するよう要求されたので，VHDLを選択する
　まだなにがなんだかわからないけれど，本を買ってこれから勉強する．
▶ライバルのAがVerilog-HDLを始めたので，私はVHDLで行く．
　ツールの購入の決裁も通った．彼には絶対負けない．
▶時流に乗り遅れたくないので，VHDLでも始めてみる
　これで，他人の見る目も変わるだろう．ところで，VHDLでゲームはできるの？（アレッ？）

こうしてみると，やはりHDLは回路設計者の人生設計における戦略的言語と言えるのかもしれません．なにはともあれ，スタートを切ろうではありませんか．

HDLの開発環境
ここでは主として，FPGA/CPLDの開発ツールを指している．初めにHDL対応を業界にアピールしたのがどのメーカーであったかは定かではないが，一社がHDL対応を謳ったのがきっかけとなり，顧客を確保することを目的としたFPGA/CPLD開発ツールのHDL対応競争が始まった．結果として1990年代末には，ほとんどのFPGA/CPLDの開発ツールがHDLに対応するようになった．その後どのような戦略の転換があったかは知らないが(HDLの導入により大規模チップの売り上げが増えたため，ツールによる収益よりもチップの販売による収益を優先させたのか，HDL対応競争の後にツールの価格競争が始まったのか？)，これらの開発ツールの価格は急激に低下し，HDLの普及に更なる拍車をかけた．

回路設計の道具
HDL評論家(？)やHDLの講師の場合は別として，一般の回路設計者にとっては，VHDLやVerilog-HDL自体は主目的ではなく，本来の目的である回路を実現するために使用する手段(道具)に過ぎない．

第2章　この言語をマスタするためのいくつかの壁

VHDLの秘密

吉澤　清

誤った性の知識が重大な結果をもたらすように，**VHDLに関する誤った理解**もまたわれわれに深刻な影響を与えます．

ここで，VHDLを誉め称え「ですから，みなさんVHDLを使いましょう」と結ぶことは簡単ですが，本書の読者諸賢にはもはや姑息な手段は通じません．

そこで，本章ではVHDLとはどういうものであるかについて，もう少し深く掘り下げて考えてみたいと思います．すぐにVHDLの書き方について読みたいという方は，まず第3章に飛んで，あとで時間が取れた折りに本章へ戻っていただければと思います．

- HDLとは人間が読みやすいように工夫されたネット・リストのようなもの
- HDLは「オブジェクト指向言語」などという生やさしいものではなく，「完全オブジェクト言語」

VHDLに関する誤った理解

VHDLをプロデュースした人々の夢や(回路設計)管理者の幻想，そして開発ツール・ベンダのセールストークなどが複雑に錯綜して，世の中へ紹介されたVHDLの虚像は，理想化されVHDLの真実とはかなり乖離したものでした．しかし，その気になれば誰しもがVHDLを使うことができるという状況に至った現在，過去のVHDLの虚像をそのまま祭り上げ続ける必要はないし，それは危険ですらある．

仮想世界の物語

昔々のことでした．知的生産世界はコンピュータ大陸に君臨する大ソフトウェア帝国とサーキット島のアナログ首長国連邦そして，ディジタル共和国からなっていました(**図2-1**)．

大ソフトウェア帝国は，巨大な国家ではありましたが，その国内では多数の異なる言語を話す民が群雄割拠し，争いが絶えませんでした．このためかどうかはわかりませんが，サーキット島は，リレー期，チューブ期，3本足期の長きにわたり大ソフトウェア帝国の支配を受けてはいませんでした．

しかし，時代が多足類期にはいると，やや状況が変わってきました．1980年頃にはディジタル共和国の一部の勢力が大ソフトウェア帝国と結び，コンピュータ大陸とサーキット島間に「**オンチップCPU**」橋を完成，孤島であったサーキット島にも大陸の文化が流入し始めました．

本来，サーキット島では象形文字に分類される「**スケマティック**」が公用語の筆記にもちいられていましたが，1990年頃になると大陸の影響を受けディジタル共和国において第2公用語としてVHDLを使用することが制定されます．このVHDLは大ソフトウェア帝国で流行していた言語を参考に作られたと言われています．

しかし，これを契機に大ソフトウェア帝国の沿岸部の勢力が言語の類似性を利して，一気にディジタル共和国の支配に乗り出したのです．これが後に語られる「**90年代の役**」です．圧倒的なソフトウェア帝国の軍事力については，サーキット島でも知らないものはないという状況でありましたから，ディジタル共和国が帝国の軍門に下るのは時間の問題であると，誰しもが思っていました．ディジタル共和国では世の終わりを信じる者たちが巷に溢れかえったのです．

ですが，世界的にインフルエンザ**Y2K型**が流行した1999年末になっても，ディジタル共和国は健在でした．昔はトグルSWでプログラムを入力していた帝国の民でしたが，近年の急速な発展にともない，贅沢な生活に慣れ，万能のOSや

オンチップCPU

半導体の集積度および製造プロセスの進化に伴い，ディジタルIC上にCPUを形成することが可能になった．最近では，CPLDの世界でもCPUの搭載は常識となりつつある．

スケマティック

Schematic．回路図のこと．

90年代の役

1990年代，HDLの登場にともない，HDLのプロデューサ達が打ち立てたコンセプトは「これからはソフトウェア技術者がHDLを使って回路設計を行う時代である」というものであった．われわれ回路設計技術者は，自分達の未来はどうなるのであろうかと不安の日々を送った．また，我々にもHDL設計はできるのではないだろうかと，虚しいとも思える努力をしたりもした．しかし，歳月が流れても我々の仕事がなくなることはなかった．いつしか勢いが良かったソフトウェア技術者達に陰りが見え，さらに暫くするとVHDLの開発環境(CPLD用)が入手しやすくなり，我々でもHDLが使えることが明らかになった．あの1990年代はいったい何だったのだろうか．

＜図2-1＞仮想世界の見取り図

ビジュアル系ソフトウェアといった社会資本なしには文字の表示さえも面倒という風潮が広まっていたのです．

このような高位レベルの設計のみに慣れた民が，過酷なサーキット島の設計環境に耐えられる筈もありません．もちろん，少数ではありますが，大ソフトウェア帝国にも苛烈な環境下でも作戦が可能なファーム族，OSドライバ・ライタといった部族がいました．しかし，彼らの多くは帝国の枢要な地位にあったため，サーキット島への侵攻に加わることはなかったのです．

結果として，ディジタル共和国は，帝国の少数の民を帰化させたのみで，帝国の全面支配はまぬがれることができたのでした．

先にも述べたように，ディジタル共和国では，言語の表記を象形文字であるスケマティックからVHDLへと切り替えることにより，国内の文化の近代化を目指していましたが，かねてより保守的なことで有名な国民は，なかなかVHDLを使おうとはしませんでした．

2002年，この事態を憂慮したディジタル共和国の文化省公用語局第2課長は，国民のVHDLに関するあまりの常識のなさに業を煮やし，これまでのゆとり教育の方針を改めるべく，国民の教育改革に乗り出したのでした．

Y2K型

2000年問題を指す．昔設計された，ソフトウェアやリアルタイム・クロック・チップが，仕様/評価の不備やゲート数をケチるために2000年以降にキチンと対応していなかったため，世紀の変わり目に起こると予言されていた，同時発生的なコンピュータ/電子機器類のトラブルのこと．対策予算を大量に投入したために問題が発生しなかったのか，それとも単なる杞憂にすぎなかったのかは誰も知らない．

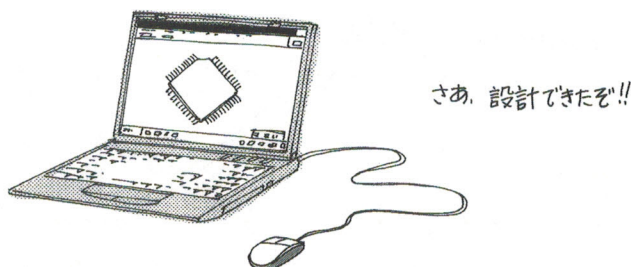
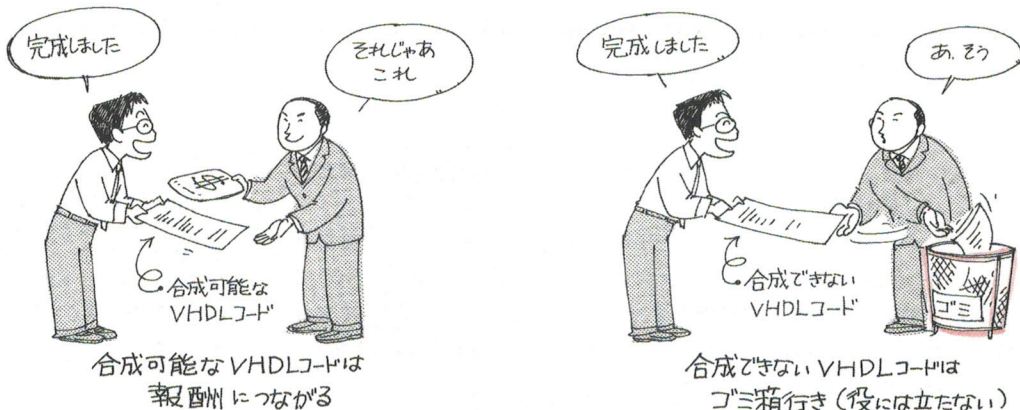
天と地の違い

すべてのVHDLコードが回路合成できるわけではありません
そこで，回路合成できるVHDLの書き方について解説しようというのが本書の主題．本書に掲載のVHDLコードの例は高度なものではないが，たいていの場合，回路合成が可能．

VHDLは回路設計用に策定された言語ですが，**すべてのVHDLコードが回路合成できるわけではありません**．

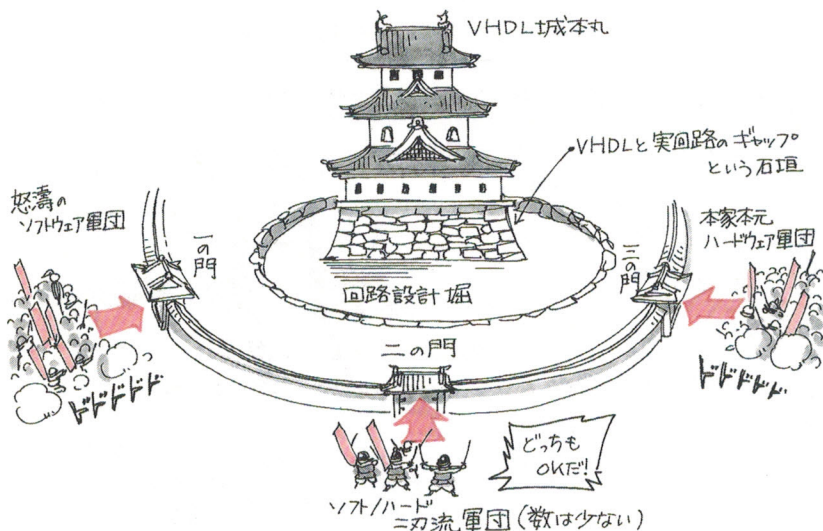
「シミュレータ上で動いてくれたのに，回路合成ができなくて」という声を時折耳にします．そうです．シミュレータ上で動いていたというのは，あくまでコンピュータ上で動いたに過ぎません．最終的に回路に落とせないのであれば，それは文字通り絵に描いた餅にすぎないのです(**図2-2**)．

価値という判断基準で見た場合，回路合成できるVHDLコードと回路合成できないVHDLコードでは，天と地の開きが生じます(**図2-3**)．実際にチップに落とすことができるコードは報酬につながるのに対して，シミュレータでしか動かないコードはゴミ箱行きです．

<図2-2>絵に描いたモチ

<図2-3>天と地と

風雲VHDL城

「90年代の役」においてもっとも熾烈をきわめたのは，VHDL城の攻略戦でした．難攻不落のVHDL城に三つの攻城軍団が総攻めをかけたのです(**図2-4**)．

一の門からは勢いに乗ったソフトウェア軍団が，二の門からは数が少ないもののソフト/ハード二刀流軍団が，三の門からは本家本元のハードウェア軍団が突

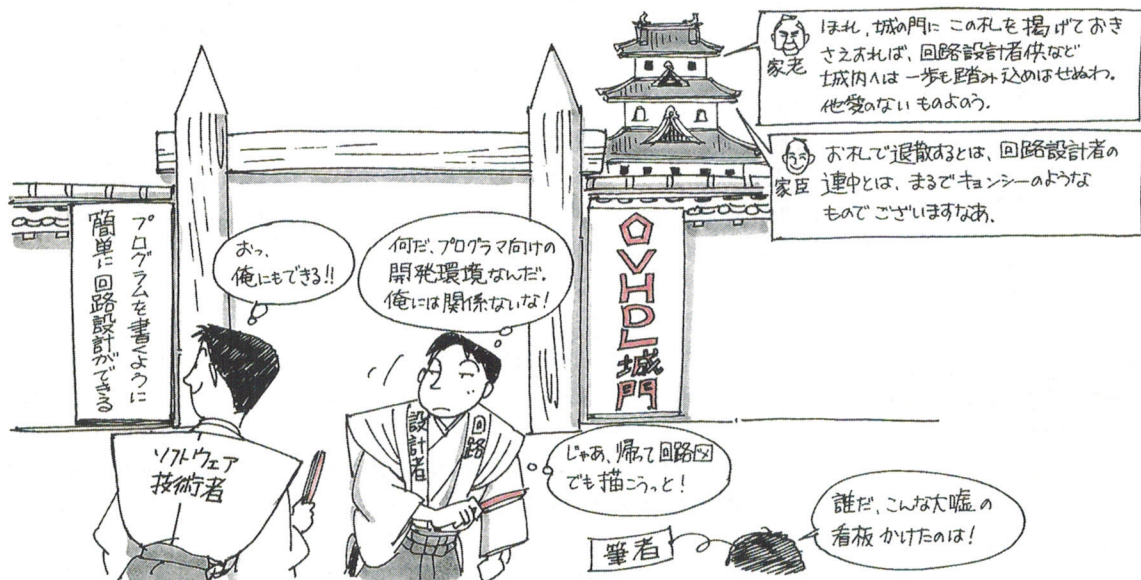
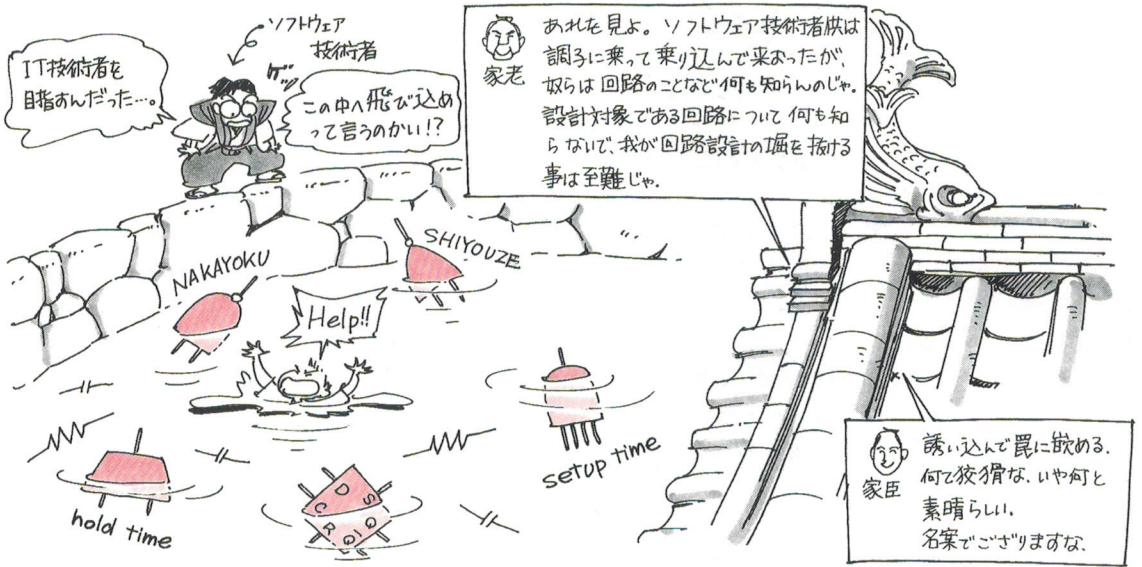
＜図2-4＞VHDL城攻城戦

入を許りました．さて，その結果やいかに．

彼らを迎え撃つVHDL城守備隊は，三つの障害を設け，城の守りを固めていました．

言霊の呪縛

みなさんは言霊の存在を信じるでしょうか．「おいおい，わたしはVHDL入門の本を買ったんだ．オカルト入門なら他所でやってくれ」．はいはい貴方の気持ちはよくわかります．けれども，事実，VHDLの世界には言霊の呪縛がかかっているのです．それは「VHDLを使えばプログラムを組むように簡単に回路設計ができる」というおなじみの文句です．VHDLの世界では常識(？)とされるこの

言霊
人が言葉に込めた思いが，ほかの人々を呪縛する状況を指す．辞書によれば，「古代，ことばがもっていると信じられていた神秘的な霊力」のこと．

＜図2-5＞回路設計者撃退バリア(第1の障害)

<図2-6>回路が掘(第2の障害…ソフトウェア技術者向けのトラップ)

文章が，実はVHDLの普及を阻害している諸悪の根源なのです．たった一行の文章に過ぎないのに，世の中に大きな影響を与えているのですから，ある意味ではすごいコピーであると言えるのですが…(**図2-5**)．

筆者が見るに，この文章は大きく二つの機能を果たしています．一つは，プログラミング経験のない，あるいは苦手とする回路設計者を，VHDLの世界からロックアウトする働きです．これまで，筆者は数人の回路設計者に「VHDLは使わないのですか」と尋ねてみたことがあります．返ってきたのは「私はプログラムは組んだことはないんだ(暗黙のうちに，だから，わたしはVHDLは使えないんだということを示している)」という判で押したような答えでした．VHDLはプログラムとは別ものであるのに，なんということでしょう．

第二の害悪は，ソフトウェア技術者をVHDLの世界へと誘い込む機能です(**図2-6**)．流石(さすが)にこの頃では，「**プログラムを組むようには回路は設計できない**」ことに気付いたのか，VHDLに手を出すソフトウェア技術者はめっきり減ったようです．それにしても，できないことをできると言い切っているこの言葉は，悪徳商法のキャッチと同じです．

ということで，本来のVHDLの使用者であるべき回路設計者を排除し，VHDL設計に対応できる筈のないソフトウェア技術者を誘惑する，この言葉がある限り，VHDLの未来は明るくありません．

本書の目的は，この言霊を無能力化することにあると言っても過言ではありません．「VHDLの入門」にいくら筆を重ねてみても，回路設計者自身が「VHDLは我々が使うために作られた道具である」という認識をもたない限り，VHDL設計者の増加率が大きく上向くことは期待できません．

わたしは声を大きくして叫びたい．「VHDLはプログラムではない」「ソフトウェア技術者のほとんどはVHDL設計には対応できない．VHDLで回路設計できるのは回路設計者である貴方だけなのだ」と．

プログラムを組むようには回路は設計できない

コンピュータのプログラムは文法通りに書けば，コンパイラのバグでもない限り，取り敢えずの動作は保証される．これは機能の豊かなコンピュータが柔軟な対応をしてくれているからにほかならない．しかし，ICチップ上の基本回路は非常に単純な働きしかできないので，コンピュータほど柔軟な対応ができるわけではない．したがって，そのあたりの回路側の事情を考えてコードを書いてやらないと，VHDLで書いてみたのだけれど回路に落ちなくてということになってしまう．残念ながら，ソフトウェア世界の常識はVHDLの世界では通用しない．

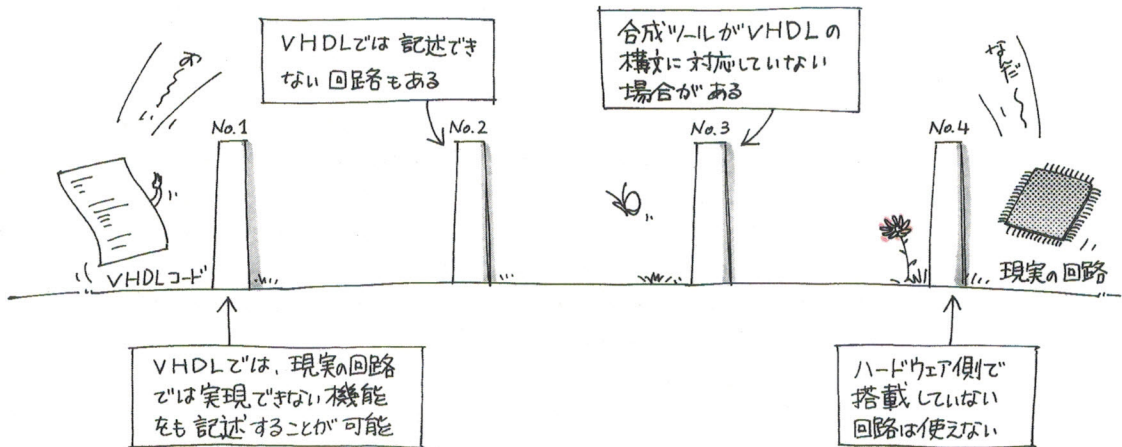
両者を隔てる非情の壁

VHDLは回路設計用の道具です．それなのになぜVHDLの言語仕様にしたがってコードを書いたのに，それが**回路合成できないなどという事態**が起こるのでしょうか．この問題を超えることができずに，入門自体を諦めた方も少なくはない筈です．

VHDLコードと現実の回路との間には，四つもの壁が存在しています(**図2-7**)．こんな状況を知ってしまうと，VHDLにより回路設計ができていることのほうが，奇跡ではないかとさえ思ってしまいます．

「VHDLでは記述できない回路」は，記述できない以上，合成ツールでNGとなることもないわけですが，それ以外の壁にぶつかると「**合成不能**」のレッドカードを頂戴するはめに陥ります．障害物競走の感を呈してきた「VHDLコードの回路合成」ですが，壁を越えるためには越えるべき壁について知らねばなりません．

回路合成できないなどという事態
驚くべきことに，VHDLの文法通りにコードを書いたとしても，そのコードが回路合成できるとは限らない．初めてこれを聞いた人は自らの耳を疑うことでしょう．それではいったいどうしたら良いのでしょうか．文法に沿っていて，かつ，回路に合成できる条件を満足するコードを書かなければならないのである．これが，VHDLの文法書は比較的やさしく見えるけれども，VHDLを使うことが難しいとされている理由である．

＜図2-7＞四つの壁

第1の壁…現実の回路では実現できない機能の記述

VHDLは，いろいろな書き方ができる**自由度の高い言語**です．しかし，自由度が高いばかりに，およそ現実の回路では実現できないような機能(動作)をも記述できてしまうことは問題です．なぜならば，そのような記述は回路合成ツールで合成不能となってしまうからです．人間が考えて実現不可能な機能の記述を，回路合成ツールに与えてみたところで実現可能にはなりません．

回路に関してなにも知らない人がVHDLで記述すると，電子回路ではどういう機能が実現できて，どういう機能が実現できないかがわかっていないわけですから，なんらかの確率でこの壁にぶつかることになります．回路規模が大きくなればなるほど，壁にあたる確率は増え，合成不能となる確率も上がります．

このような事態を回避するためには，やはり回路設計の経験をもった人間がVHDLコードを書く必要があります．もちろん，回路についてはなにも知らない人間が試行錯誤の結果として合成可能なコードを書けるようになることも不可能ではないかもしれませんが，それには膨大な時間がかかりそうです．合成の可，

合成不能
VHDLコードを回路合成ツールが解析した結果，そのコードのもつ機能を現実の回路に展開できないと判断された場合には，回路合成ツールは合成不能を呈示する．

自由度の高い言語
この場合には，実現したい処理内容に対して，その記述の仕方がいくつも存在する状況をさす．

不可が実際の回路の振る舞いに左右される以上，回路設計の体系を利用したほうが，まったくランダムな試行と比較して効率が上がるという道理です．

第2の壁…VHDLでは記述できない回路もある

シュミット・トリガ
入出力電圧特性がヒステリシスをもつ回路．主として入力信号のノイズの除去や，簡単な発振回路などに用いられる．

プルアップ
プルアップ抵抗．ハイ・インピーダンス状態となり得る信号ラインの電位を安定させるために，抵抗を介して十側の電源電圧に接続する処置．ICの入力ピン，Nチャネル・オープン・ドレイン出力，3ステート出力などに対して使用する．

いかにVHDLが自由度の高い言語であるとは言っても，やはりVHDLも人の作りしモノなので，そこには限界があります．VHDLでも表現できない回路というものがやはり存在するのです．当然のことながら，表現できない回路というものは，VHDLの反映として実現することはできません．

このような例としては，**シュミット・トリガ**，**プルアップ**，**プルダウン**，**双方向性のアナログSW**などがあげられます．

VHDLにおいては，データのレベルは"1"と"0"の2値で表現されます．"1"と"0"の間の中間的な電圧レベルについては，細かく表現することはできません．このため，出力が入力の中間的な二つの電圧レベルで変化することを特徴とするシュミット・トリガ回路の働きをVHDLでモデリングすることはできません．

プルアップ抵抗やプルダウン抵抗を記述しようとする場合に問題となるのは，VHDLにおいては二つ以上の信号源が一つの信号(配線)を同時に駆動した場合の取り扱いが，実回路の動作を完全に反映してはいない点です．

たとえば，プルアップ抵抗側が"1"を出力し，N-chオープン・ドレイン出力が

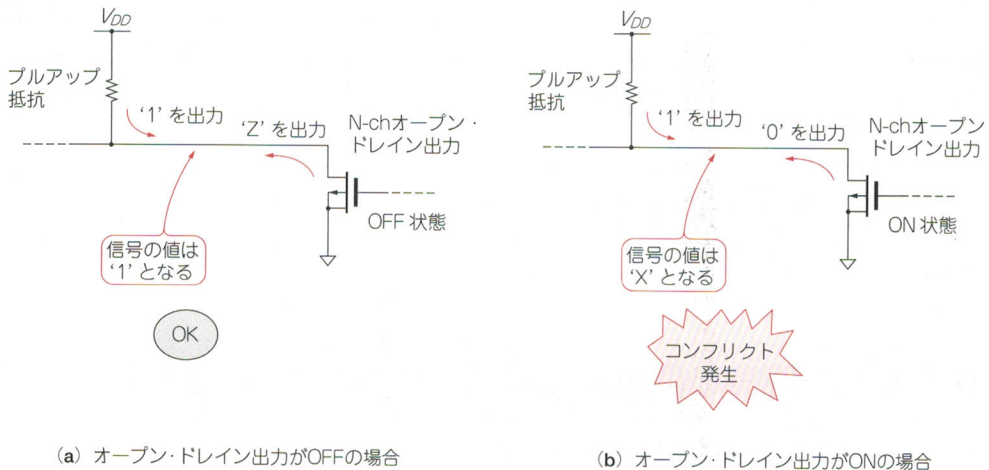
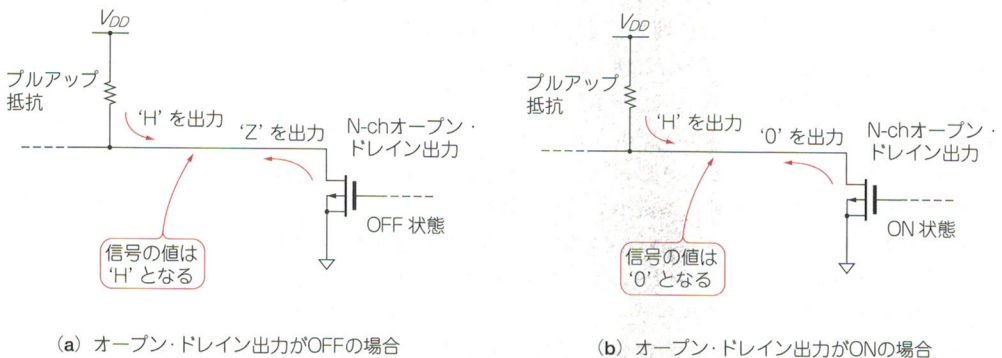
<図2-8>プルアップ抵抗とオープン・ドレイン出力（Ⅰ）

(a) オープン･ドレイン出力がOFFの場合

(b) オープン･ドレイン出力がONの場合

<図2-9>プルアップ抵抗とオープン・ドレイン出力（Ⅱ）

(a) オープン･ドレイン出力がOFFの場合

(b) オープン･ドレイン出力がONの場合

“Z”または“0”を出力するとします(図2-8).

この場合，プルアップ側の出力が“1”，オープン・ドレイン側の出力が“0”のときには，信号(配線)の値が“X”(コンフリクト発生)となってしまい，現実の回路の動作とは一致しません.

また，VHDLの信号レベルの“H”(弱い信号の“1”)という概念を用いて，プルアップ側の出力を“H”とした場合には，図2-9のような動作となります.

今度はコンフリクトは起こりませんが，本来信号レベルを“1”としたいところが“H”(弱い信号の“1”)となってしまいます．この結果をほかの回路で使う場合には，信号の“H”という状態を“1”の状態に変換するバッファを介する必要があります.

社内での検討の際には,便宜上このような方法も使うことができるでしょうが,公にこのような方法を採ってよいかどうかについては疑問が残ります.

双方向性のアナログSWは，単に制御信号の値により配線の間が繋がったり切れたりする回路にすぎません．しかし，VHDLでこのような回路の記述を行うことは至難の技(事実上不可能)です．VHDLには複数の配線(信号)を繋いだり切り離したりという概念がありません．また，双方向にバッファ的な回路を接続する形で表現をしようとすると，片方の回路の出力がもう片方の回路の入力に影響を与え，ラッチのようになってしまい，外界からの状態に反応することができなくなります.

プルダウン
プルダウン抵抗．ハイ・インピーダンス状態となり得る信号ラインの電位を安定させるために，抵抗を介してGND電圧に接続する処置．ICの入力ピン，Pチャネル・オープン・ドレイン出力，3ステート出力などに対して使用する.

双方向性のアナログSW
C-MOSのアナログSWは双方向に信号の伝達を行うことができる.

バッファ
論理レベルが反転しない緩衝増幅器のこと．レベル変換機能を伴うことがある.

第3の壁…合成ツールがVHDLの構文に対応していないことがある

もうすでに完璧の域に達している回路合成ツールも存在するかもしれませんが，現存する回路合成ツールの多くには，なんらかの制限や対応できていない記述の仕方などが存在します.

この原因もまた，VHDLの言語としての自由度にあります．自由度が高いということは一見よいことのように見えます．設計者である貴方にとっては，記述の仕方に関して自由度が高いほうが便利かもしれません.

でも，回路合成ツールの立場にもなってください．一つの回路に対して，二つも三つも書き方があるとしたら，回路合成の作業はたいへんです．もちろん，そのたいへんさが回路合成ツールの存在理由でもあるわけですから，悪いとばかりも言えないのですが.

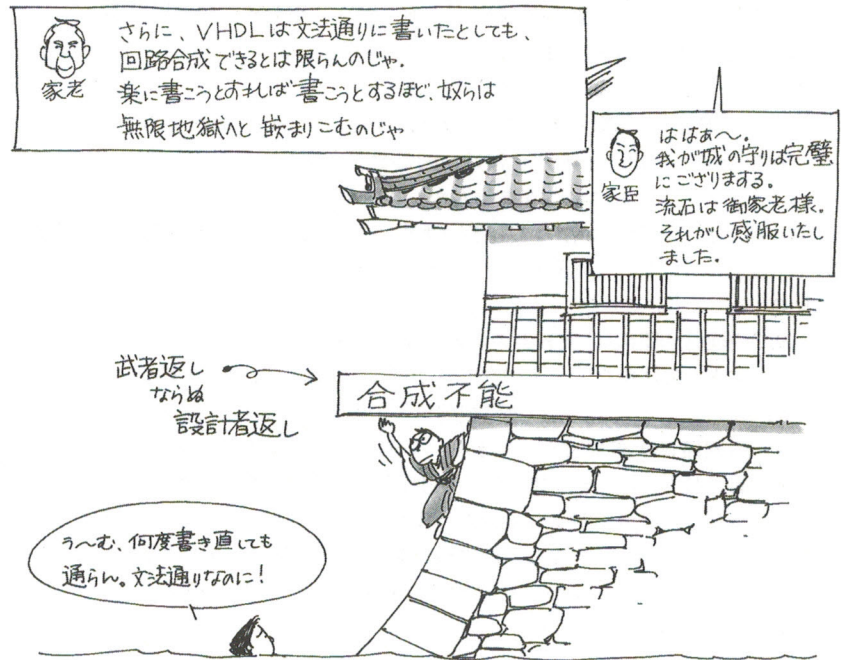
回路図設計の時代は，回路合成ツールがかならずしも必要とはかぎりませんでした．回路図上のシンボルが，実際の要素回路と一対一で対応していたので，単純にシンボルを要素回路にアサインすることで事が足りたのです.

しかし，VHDLの場合はいろいろな書き方ができます．したがって，現実の回路と一対一で対応しているとはいえません．このため，VHDLで記述された内容と現実の回路の対応付けをしてやるという作業が必要になります．これを行うために生まれたのが回路合成ツールというわけなのです.

今日においては，VHDLは電子業界に広く浸透していますが，その歴史は短い電子回路の歴史の中でももっとも短いものと言えるでしょう．実際にVHDLによる設計が一般化したのはごく最近，1990年代に入ってからです．このような背景から見てもわかるように，現在使用されている回路合成ツールの多くは数年～十年程度の年月しか重ねていません．つまり，大概の回路合成ツールは成熟の段階にいたる発展段階にあるというわけです.

このため，貴方が使おうとしている合成ツールが完全であるという保証はあり

回路合成ツール
VHDLコードを読んで，そのコードが表している機能を解析し，その機能を現実の回路の上に再構成する．これが回路合成ツールの機能.

＜図2-10＞そして無限地獄（第3の障害）

ません．そして，合成ツールによって，そのできは異なります．しかし，なにが合成できて，なにが合成できないかが，実際に回路合成をかけてみるまでわからないという状況は困りものです．とくに初めてVHDLに入門しようとする場合に，この構文は回路に落ちるのだろうかと，回路合成ツールに不安（不信）と疑念をもっていたのでは，VHDLの習得どころの話ではないでしょう．

現実の回路の動作に近い
「ディジタルの基本回路により，容易に構成ができる」というような意味．ディジタルの基本回路により容易に再構成が可能な記述であれば，回路合成が行えることはほぼ確実と言える．

回路合成ツールで合成可能かどうかの目安が一つだけあります．それは，合成しようとするコードが**現実の回路の動作に近い**か，それともそれからかけ離れているかというファクタです．現実のディジタル回路は，ゲートやフリップフロップなど非常にプリミティブな要素により構成されます．このためVHDLの構文のうち，プリミティブなものに関しては，概ね合成が可能であるという傾向があります．つまりは，現実の回路より乖離した高級な記述ほど，合成不能に陥る可能性は高いということができます（**図2-10**）．

なんのことはありません．結論は，「VHDLは簡単（プリミティブ）な構文から始めよう」というごくあたりまえのお話となってしまいました．くれぐれも「VHDL，合成できなければただのゴミ」という大原則をお忘れなきよう．

「〜の本」に載っていたVHDLコードが合成できなかった，などという局面においては，その本で使用していた回路合成ツールがなんであったか，ということもキーポイントになるので，チェックをしてみることをお勧めします．

第4の壁…ハードウェア側で搭載していない回路は使えない

要は「ない袖は振れない」ということです．貴方が使おうとしているデバイスには搭載されていない回路や機能をVHDLで記述できるからといって使おうとすると，当然のことながら合成不能という答えが返ってきます．

もし，それがどうしても必要な回路や機能であるならば，なにか工夫をして使

っているデバイス上で同等の動作をさせるとか，チップ上で実現できない部分を**ディスクリート部品**を使って外付けにするとか，使用するデバイスを変更するなどして対処しなくてはなりません．

これら，VHDLと現実の回路を隔てている壁を突破するための武器は，「回路に関する知識」だけです．そして，その知識を持っているのは，回路設計者たる貴方なのです．

ディスクリート部品
通常は，個別半導体(トランジスタやダイオードなど)や*CR*類などを指す．しかし，この場合は汎用のICチップなど，被開発チップ以外のICチップまでを含んでいる．

ソフトウェアとハードウェアの開発環境の大きな差

大昔(とは言っても30年くらい前の時代)，計算機本体のコンソールSWをパチパチ操作してコンピュータをプログラムしていた時代はともかく，現在のソフトウェアの開発環境はたいへん豪華なものとなっています．

まず，コンピュータ．数万から数十万ゲートのハードウェアの固まりです．処理速度は高速で，大容量のメモリとハードディスクを搭載し，インターフェース(キーボード，マウス，ディスプレイ)やペリフェラルも充実しています．

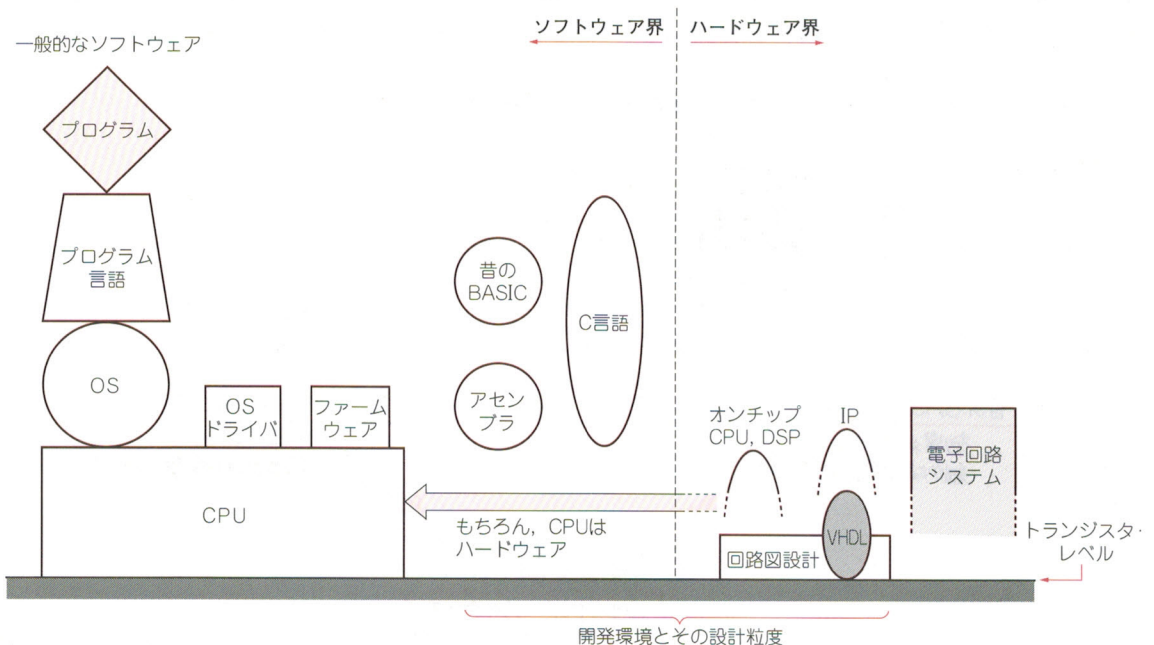
そしてOSの存在．プログラムがハードウェア(コンピュータ)に直接アクセスするだなんて信じられません．外界や2次記憶とのインターフェースから主記憶の管理まで，プログラムの走行を支援する**豪華なルーチン群**が貴方のプログラム/プログラム開発環境をサポートします．

さらにはプログラム用の高級言語．種類は豊富．アプリケーションと貴方の好みに応じてベストフィットなものを選び放題．高機能なコマンド群，そしてきめ細かな処理を可能とするコマンド群が貴方の成功を約束しています．

いやぁ，なんだか高級すぎて使いこなすのがたいへんそう…．いえいえ，昨今の顧客の要求に応えるためには必要なことなのです．

翻って，ディジタル回路の設計現場はというと，相変わらずゲートやフリップ

豪華なルーチン群
コンピュータのオペレーティング・システムには，記憶（メモリやディスク）管理や表示画面とのインターフェース（図形描画やウインドウ描画），そして周辺装置のインターフェースを取り扱うための，ありとあらゆるルーチン(サブルーチンまたはそれに類する個別プログラム）が用意されている．

<図2-11>ソフトウェアとハードウェアの設計基盤の違い

設計粒度と記述の細やかさ
少ない行数で多くの機能を記述することと，機能について微に入り細にわたって記述することとは，両立することができない．

フロップをベースとした設計，VHDLが導入されたとはいえ，**設計粒度と記述の細やかさ**はトレードオフの関係にあるため，設計の全般にわたって高い設計効率を保つことはけっこうむずかしい．回路シミュレータが入手しやすくなって，実機を組まなくても回路の動作を見ることができるようになり，内蔵CPUも一般化してはきたけれど…(**図2-11**)．

こうやって，ソフトウェアとハードウェアの開発環境をくらべてみると，かなり大きな差があることがわかります．このように，大きな差異が存在し，必要な基礎知識も異なるため，両者の間の移行は容易なモノではないといえます．

プリミティブな記述の習得優先論

高度な記述
これが高度な記述だと一概に定義することは難しいが，一般的には機能記述の中でも，設計効率の高い書き方のことを指しているもよう．

VHDLの教科書にはかならず「高度な記述ができるようになろう」とカッコよい文句が登場します．しかし，「**高度な記述**」とはいったいなんなのでしょう．

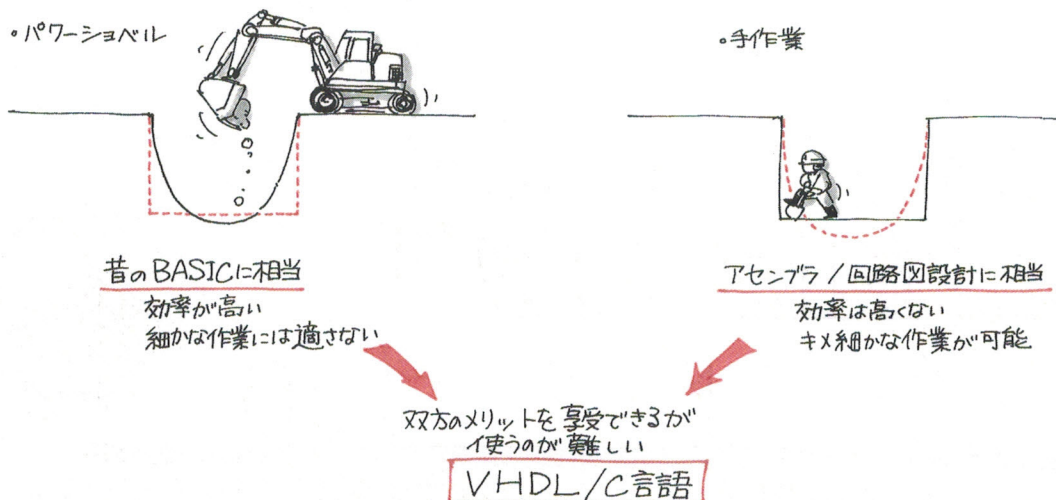
①プログラムのように書くことが高度な記述なのでしょうか
②わずかな行数で膨大な回路を生成できるような記述が高度なのでしょうか
③それとも，プリミティブなレベルから構造を積み上げて築いた巨大なブロックを高度というのでしょうか

どうやら文脈から判断するに，プログラム的に書けて設計効率が高い書き方を「高度な記述」と称しているようです．

しかし，筆者は，この「高度な記述」推進論には懐疑的です．VHDLと同じく難しいといわれている言語にCがあります．そもそもC言語が難しいと言われるのは高度な記述，言い換えると設計効率の高い記述ができるからだけではないはずです．効率のことだけを考えるなら，昔のBASICだってちょっとしたものだったでしょう．C言語が難しいと言われるのは，開発効率の高い記述が可能であると同時に，**アセンブラのレベルに近い細かな記述**をも可能にしているところにあります．

アセンブラのレベルに近い細やかな記述
アセンブラ言語によれば，コンピュータの機械語レベル，つまりコンピュータにおけるいちばん粒度の小さな操作の単位で記述することができる．C言語によれば，それにかなり近いレベルの記述にも対応することが可能．

たとえがよくないかもしれませんが，これを土木工事にたとえてみましょう．パワーショベルは効率の高い道具であり，一気に大量の土砂を掘り起こすことが可能です．しかし，反面，センチ単位の細かな作業が必要な場合はやはり手作業

<図2-12>パワーショベルと手作業

に頼らざるを得ません．これをたとえるならば，アセンブラというところでしょう（図2-12）．

それでは，VHDLやC言語にあたるモノとはいったいなんでしょうか．残念ながら，そんな道具が存在するという話は聞いたことはありません．しかし，そんな道具がもし存在するとしたら，おそらく漫画家の松本零士氏がデザインするような超未来的なスーパーメカになることでしょう．ユーザ・インターフェースがどのようなものになるかはわかりませんが，コンピュータ化したとしても，人手による作業よりはおそらく複雑でしょう．二つの背反する要求を満足した結果として操作は複雑となるわけです．

同じように，VHDLやC言語が難しいと言われるのも，開発効率の向上と細かい操作という背反する二つの要求を満足した結果，そのツケが回路設計者やプログラマにまわってきたわけです．ただし，これはかならずしも悪いことではありません．二つの要求を満足する言語がないよりは，使い方が多少難しくとも，そういった言語が存在しているほうが，設計者にとって自由度が高いと言えるからです．

さてそれでは，VHDLにおいて優先度が高いのは**プリミティブな記述**でしょうか，それとも「高度な記述」でしょうか．たとえば，貴方がクライアントからの要求に応じて回路を設計する場合を想像してみてください．「高度な記述」を駆使してあなたは回路を作り上げました．しかし，何度オプションを変更してコンパイルしてみても，細かな部分が仕様通りにはならなかったとします．そんな場合，あなたのクライアントや上司はどう思うでしょうか．「そんな君，開発効率が上がったのだから，たかが仕様の一部が実現できなかったことぐらい気にすることはないよ」と応じてくれるでしょうか．

プリミティブな記述
Primitiveは，原始的あるいは根本のの意味．ここでは，ディジタル回路の基本回路に近いレベルで記述することを指す．

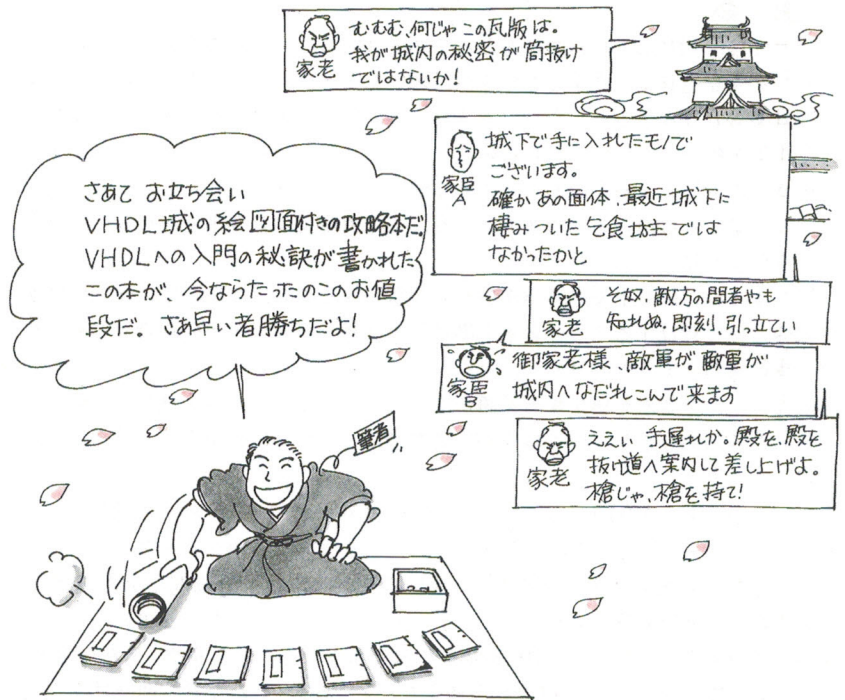
場合によってはそれで通ることがあるかもしれませんが，仕様がわずかに異なったためにインターフェースが破綻をきたしたり，装置が異常な動作をするなどということは枚挙にいとまがありません．世の中ではこのような状況を未完成と呼びます．

プリミティブな記述によれば，開発効率は多少低下するものの，ディジタル回路で実現可能な回路であるならば，概ね実現することは可能です．このような状況から，筆者は「高度な記述」よりもプリミティブな記述の習得を優先すべきであると考えます．

＜図2-13＞How to瓦版

コンセプトの陳腐化

VHDLは難しいね

VHDLは使い方を憶えるのが難しい言語です．そして，解説書の多くがプログラムを組める人向けに書かれていることが，回路設計者がVHDLへアプローチする敷居をさらに高くしている．本書では，このような状況を打開すべく，本当に必要な構文のみを厳選し，各構文の解説にページを割き，誰でも使うことができる開発ツールでコンパイル可能な，サンプル・コードを掲載してみた．

さて，これまで述べてきたように，ソフトウェア技術者がハードウェアの設計を行おうとする場合には，大きな困難が予想されます．ソフトウェア世界においてVHDLフィーバーがピークを迎えた1990年代においてさえも，ハードウェア設計への移行を果たしたプログラマが希有であったことは，これを裏打ちしています．

しかも，バブル崩壊直後のプログラマ過剰の時代はとうに終わり，IT関連において需要が逼迫している昨今，ソフトウェア技術者をハードウェア設計にシフトさせようというコンセプトそのものが根底より崩れつつあります．

だいたいにして，進歩の速いこの業界において，20年近くも同じコンセプトを尊守しながら，「VHDLは難しいね」と言って笑っている．なにか間違っているんじゃないかとか感じたりはしないものなのでしょうか．

せっかく我々の目の前にはVHDLというすばらしい道具があるのです．現在の閉塞した状況は打開されねばなりません．そこで登場するのが本書のコンセプトです．本来，このような提言を一介のエンジニアに過ぎない筆者がすべきではないのかもしれません．しかし，すでにVHDLに関する考え方を変えなければならなかった時期というのは過ぎてしまっているのです．ここで提示するコンセプトは完全ではないかもしれません．ですが，まずは時代にそぐわなくなってしまった旧いコンセプトを捨て去らなくてはなりません．そうでなければ，VHDL設計技術者の不足の解消どころか，不足が助長されかねないのが現在の状況なのです(**図2-13**)．

本書のコンセプト

本書におけるVHDLに関する考え方は，以下のようなものです．

- VHDLを使うと，ソフトウェア技術者がプログラムを組むように簡単に回路設計できる，などということはない
- 回路設計に関する知識がないと，回路合成可能なVHDLコードは書けない
- VHDLを使って回路設計の実務ができるのは99％回路設計者である
- 経験のある回路設計者は，ゲートとフリップフロップの書き方がわかり，階層設計の方法を知っていれば，ほとんどの回路を設計することができる
- とはいえ，VHDL記述をする上で便利な機能（ファンクション，他）は使えるに越したことはない
- 教科書でいう「高度な記述」は回路設計者には読みづらく，言われているほど能率が上がるとは限らず，また，回路合成ではNGとなる可能性も高い

このような考え方をふまえ，本書は次のようなコンセプトに基づいて作られました．

- プログラマ向けではなく，回路設計者フレンドリな内容を心掛ける
- VHDLシミュレータ上のみで働くコードではなく，回路合成が可能なコードの書き方について解説をする
- 掲載するVHDLコードの記述例は，アルテラ社のMAX＋plus IIで直接回路合成が可能なものとする（ほとんどのコードはサイプレス社のWarp2にも対応）
- CPLDをターゲット・デバイスとした製作例を添付する
  実チップへのインプリメント例を実践に即したサンプルとして取り上げる
- 実用性の高いサンプル・コードを掲載する

さあ，いつまでも文法書を片手にシミュレータ上で戯れていても，なんの問題解決にもなりません．取り敢えずCPLDを使って実際の回路上でVHDLコードを働かせてみましょう．当然，いろいろな問題も持ち上がってくるでしょうが，回路設計者の貴方が腰を落ち着けて取り組めば，解決を図ることはできるはずです．

VHDL設計も，習うより慣れろです．まずは始めること．ただし，最初はくれぐれも簡単な回路からゆっくりとです．そして，VHDLによる設計をエンジョイしていきましょう．

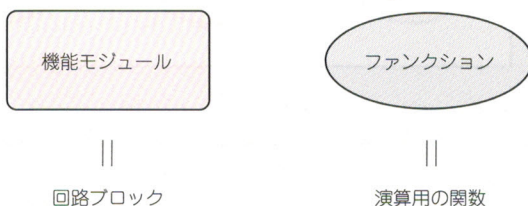
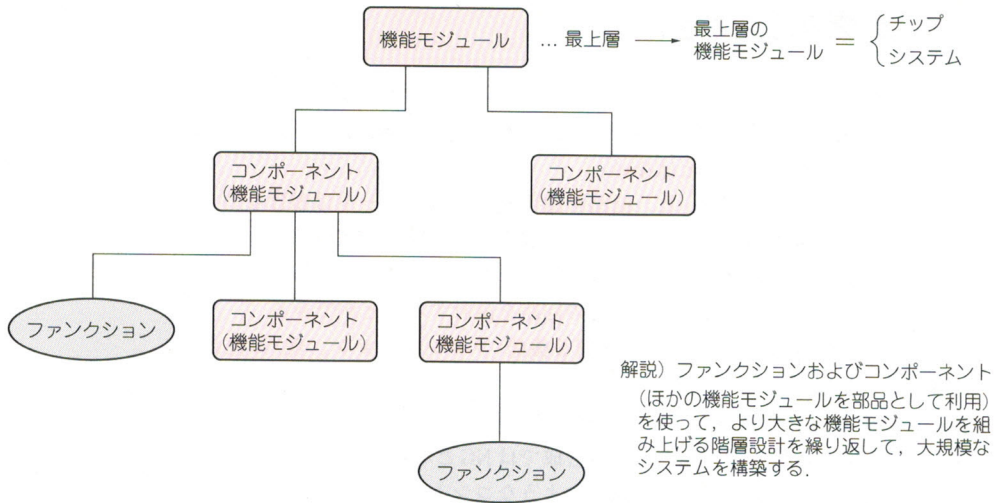
CPLD
Complex Programmable Logic Deviceの略．構造的には，1チップにPLDをたくさん集積して，内部で自由に相互配線ができるようにしたもの．外から見ると，回路データを書き込むことにより，内部回路が変更可能なディジタルIC．

VHDLの構造

本書で解説するVHDLの構成単位は，機能モジュールとファンクションの二

＜図2-14＞VHDLを構成する二つの要素

＜図2-15＞VHDLの構造

解説）ファンクションおよびコンポーネント（ほかの機能モジュールを部品として利用）を使って，より大きな機能モジュールを組み上げる階層設計を繰り返して，大規模なシステムを構築する．

つです（**図2-14**）．

機能モジュールは回路図設計の場合の回路ブロックに相当します．機能モジュールの中ではほかの機能モジュールをコンポーネント（部品）として使うことが可能です．これにより，**階層設計**が実現できます（**図2-15**）．

ファンクションは，機能モジュールの記述に使用可能な関数であり，単一または複数のデータを与えることにより，それに基づいた演算結果を返します．VHDLにおいては，標準的なデータ上におけるプリミティブな演算が組み込み関数として十分に整備されていないため，あらかじめ必要な演算などについてファンクションを用意しておくと，実際にコードを書くおりに重宝します．

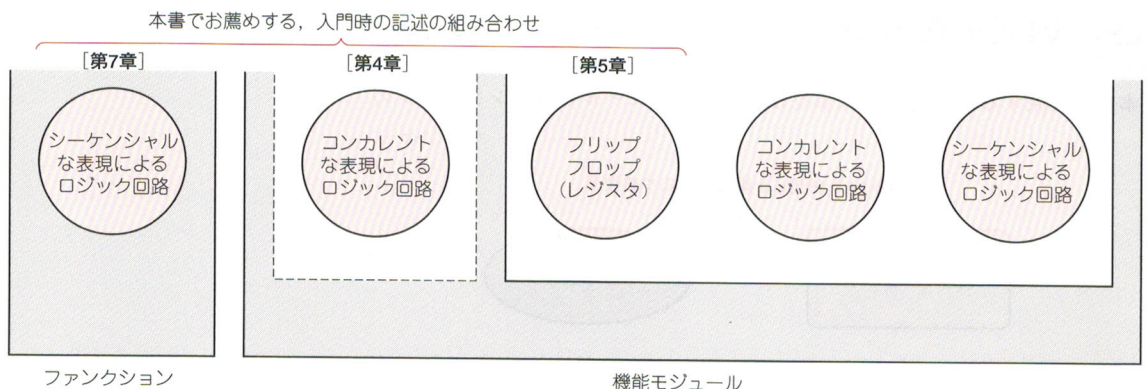
VHDLにおいては，大きく分けて

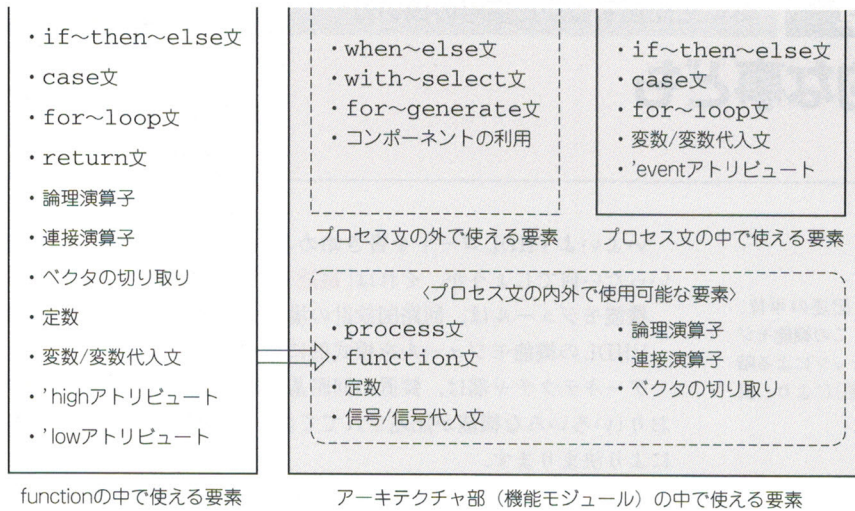
- コンカレントな表現によるロジック回路の記述
- シーケンシャルな表現によるロジック回路の記述
- フリップフロップ（レジスタ）の記述

の3通りの記述が可能です．ただし，VHDLコードの全域で，すべての表現が可能なわけではありません．**図2-16**にVHDLの記述領域と記述できる内容の対応

階層設計

積み木のように，小さなブロックを積み重ねて大きなシステムを組み上げていく設計法のこと．特別なことではなく，知らず知らずのうちに誰もが行っている．なぜなら，大きなシステムを一気に組み上げることは難しく，階層設計をしたほうが設計が容易となるから．

＜図2-16＞記述領域と記述できる内容

注）大かっこ内は，それぞれの回路の記述について解説している章を示す．

＜図2-17＞記述領域と使うことができる要素

を示します．

ファンクションの中では，シーケンシャルな表現によるロジック回路の記述のみが可能です．機能モジュールの内部は，さらにプロセス文により記述可能な内容が異なる二つの領域に分割されます．プロセス文の外ではコンカレントな表現によるロジック回路の記述のみが可能です．そして，プロセス文の中においては，コンカレントな表現によるロジック回路の記述，シーケンシャルな表現によるロジック回路の記述，フリップフロップの記述のすべてが可能です．

「そうか，それならば機能モジュールの**プロセス文**の中で，すべてを記述すればよいわけだね」とおっしゃるそこの貴方．ちょっと待ってください．同一領域内ですべての記述ができるということは，**両刃の剣**でもあるのです．機能モジュールのプロセス文の中で，すべての記述が可能という事実は，一見便利そうです．しかし，3通りの記述がたがいに似通っていて，同じ構文により構成されているとしたら，そして，微妙な記述の違いがそれぞれの記述の違いを決定づけているとしたら，それは混同や間違いが起こりやすく，記述が難しいと言うことにほかなりません．

プロセス文
VHDLにおける主要な構文の一つ．万能の記述領域を確保することができるが，なんでもかんでも詰め込むことができるため，かえって，すべての機能を使いこなすことがたいへんになっている．このため本書では，はじめはプロセス文をフリップフロップ(レジスタ)の記述に限定して使用してみることを提案している．

両刃の剣
使い方により，利にもなるが害にもなるさまを表している．

このため，本書では，少なくとも最初は，できるだけ書き方の異なるおたがいの境界線がはっきりした記述の組み合わせを使って，入門を始められることをお勧めします．

つまり，それぞれの記述を，

●コンカレントな表現によるロジック回路の記述　…機能モジュールのプロセス文の外
●シーケンシャルな表現によるロジック回路の記述…ファンクション
●フリップフロップ(レジスタ)の記述　…機能モジュールのプロセス文の中

というように記述領域を変えて行うわけです．

最後に，VHDLの各記述領域において，記述に使うことができる要素について**図2-17**にまとめておきます．ただし，各記述領域を形成するフレームとなる構文(entity文やarchitectur文など)については割愛してます．

第3章　なにを記述するかと信号の扱い方

基本的な事ども

吉澤　清

いよいよVHDLコードを書き始めるわけですが，VHDLで記述するものとはいったい何でしょうか．それは「**機能モジュール**」です．

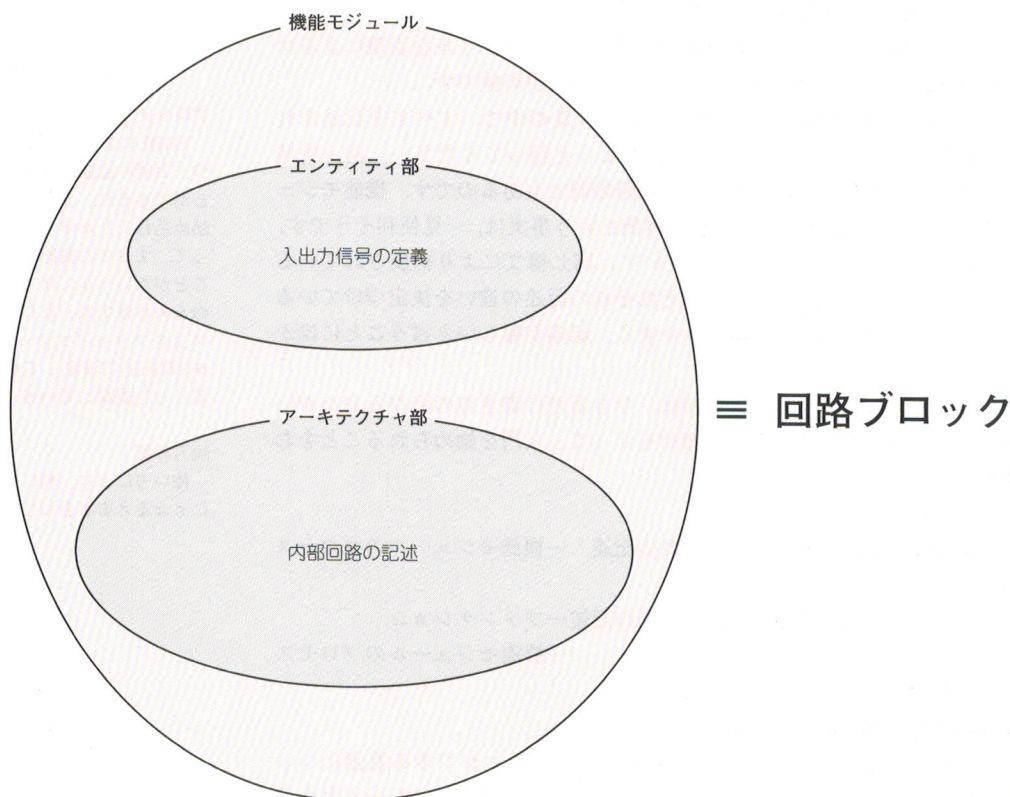
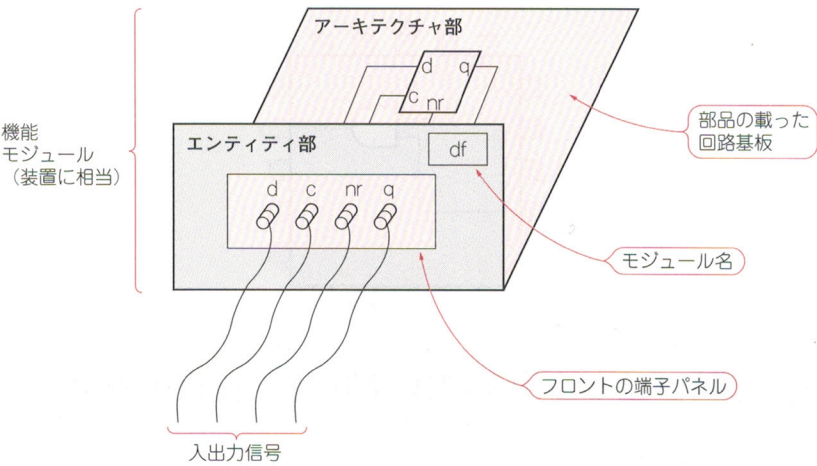
機能モジュール
VHDLにおける記述の単位．VHDLにおいては，この機能モジュールとファンクションによる階層設計(第6章を参照)により，回路を組みたてていく．

機能モジュールは，回路図設計の場合の回路ブロックに相当します(**図3-1**)．

VHDLの機能モジュールを模式的に描くと，**図3-2**のようになります．

アーキテクチャ部は，装置の回路基板に相当します．いろいろな部品が載っており(いろいろな機能が記述されてており)，装置(機能モジュール)の動作はこれにより決まります．

エンティティ部は，装置の端子パネルに相当します．機能モジュールはエンティティ部で定義された入出力信号を介して外界とやりとりをします．IC向けの最上位の機能モジュールにおいては，エンティティ部に定義された信号がそのままチップの入出力ピンに対応することになります．

〈**図3-1**〉**VHDL記述の構造**(機能モジュールの構造)

〈図3-2〉機能モジュールの構造の例

記述するのは「機能モジュール」である

VHDLコードはテキスト記述なので，ちょっと見にはコンピュータのプログラムのようにも見えます．しかし，VHDLで書くことができるのは「**回路**」であってプログラムではありません．

表3-1にANDゲート1個を含んだ機能モジュール“andLogic”のVHDLコードと実際に記述されている回路を示します．

両者を照らし合わせると，VHDL記述の雰囲気が少しはつかめると思います．

このような簡単なVHDLコードでも，これだけで完結しており，合成ツールで回路合成を行い，得られた**コンフィギュレーション・コード**を**CPLD**に書き込めば，立派にANDゲートとして働くチップが得られます(**図3-3**)．

このように，「VHDLで回路を設計する」こととは，自分の必要とする機能をもった機能モジュールを記述することにほかなりません．

回路

VHDLは，主としてディジタル回路の構造/機能の記述に使われる．

コンフィギュレーション・コード

Configuration Code. Configurationは配置とか構成の意．プログラマブルなCPLDやFPGAの内部構造を決定するデータのこと．CPLD/FPGAの開発ツールは，回路合成の結果を基に，CPLDやFPGA用のコンフィギュレーション・コードを生成する．

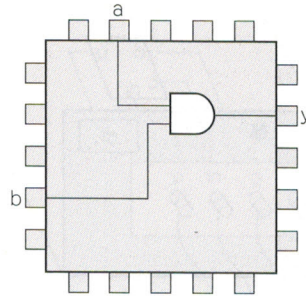
CPLD

Complex Programmable Logic Device. プログラムすることにより内部回路の機能が変更可能なディジタルIC．本書ではアルテラ社のMAXシリーズとFLEXシリーズのCPLDをターゲット・デバイスとして使用する．

〈表3-1〉機能モジュールの例(ANDゲートの記述)

VHDLコード	記述されている回路

--
-- VHDL code of AND gate
--
library ieee;
use ieee.std_logic_1164.all;
 	
entity andLogic is
  port(a : in std_logic;
       b : in std_logic;
       y : out std_logic);
end andLogic;
 	
architecture rtl of andLogic is
  signal notUse0 : std_logic;
begin
  y <= a and b;
end rtl;
	andLogicモジュール
	a
	b
	y

〈図3-3〉表3-1のVHDLコードをCPLDにインプリメントすると…

機能モジュールと回路ブロックを対比して(VHDL記述の構造)

一般的に回路図設計においては，回路ブロックを表現するために，回路図(回路ブロックの内部回路)と，回路ブロックにアクセスするためのシンボルを使います(**表3-2**).

それでは“andLogic”機能モジュール/回路ブロックの場合を例にとって，VHDLコードと「シンボル」，「回路図」を比較しながら解説を進めます．

表3-2の右側のVHDLコードを参照してください．

●コメント

初めの3行はコメントです．VHDLコードを書く際に，モジュールのタイトルや説明などを付けておくと，あとでコードを参照するときに便利です．VHDLでは“--”(ハイフン・ハイフン)を書くと，その位置から行の右端までがコメントとして取り扱われます．

●ライブラリの使用宣言

コメント行に続く2行では，ライブラリを使用する旨の宣言を行っています．

〈表3-2〉機能モジュールと回路ブロックの対比

回路図設計(回路ブロックの記述)	VHDL設計(機能モジュールの記述)
・シンボル andLogic a b y ・回路図 a b y	下記コード参照

```
--
-- VHDL code of AND gate
--
library ieee;
use ieee.std_logic_1164.all;
entity andLogic is
  port(a : in std_logic;
       b : in std_logic;
       y : out std_logic);
end andLogic;
architecture rtl of andLogic is
  signal notUse0  : std_logic;
begin
  y <= a and b;
end rtl;
```

- コメント(タイトル)
- ライブラリ使用の宣言
- エンティティ部 ― 回路図設計の場合の「シンボル」に相当する部分
  - 入力信号の定義
  - 出力信号の定義
- アーキテクチャ部 ― 回路図設計の場合の「回路図」に相当する部分
  - 内部信号の定義
  - 実際の回路の記述

▨の位置にモジュール名を記入する

ここでは，IEEEライブラリ群の中の"std_logic_1164"というライブラリを使用することを宣言しています

これは，VHDLで標準的に使われる"std_logic"型と"std_logic_vector"型を使うために必要となるライブラリです．

本書では，データ型としてこれら二つの型を使用するので，この2行の記述は必須となります(モジュールごとに宣言することが必要)．

IEEEライブラリ群
IEEEで標準化された，VHDL用のライブラリ(複数)のこと．IEEEは，The Institute of Electrical and Electronics Engineers, inc. の略で，エレクトロニクス関連の米国の学会のこと．

●モジュール名を書く位置

表3-2のVHDLコードに3箇所アミがかかっている部分があります．

機能モジュールを記述する場合，この3箇所にモジュール名を書かなければなりません．三つのモジュール名が一致していないとエラーになるので，モジュール名を変更する場合には注意してください．

モジュール名
VHDLで記述する各機能モジュールには，識別用の固有のモジュール名をつける．

●エンティティ部

機能モジュールのエンティティ部が回路図設計の場合の「シンボル」に相当します．エンティティ部では，モジュールの入出力信号の名前と方向性，信号の型の指定を行います．

表3-2の記述では，たとえば信号aは入力信号で，std_logic型です．信号yは出力信号で，やはりstd_logic型です．

●アーキテクチャ部

機能モジュールのアーキテクチャ部が，回路設計の場合の「回路図」に相当します．

アーキテクチャ部では，rtlというキーワードが2回出てきます．これは，アーキテクチャ部で記述する記述レベルがRTLレベルであることを示します．

VHDLでは，一つの機能モジュールに対し複数のアーキテクチャ部の記述が許されており，この位置に書かれるキーワードは本来複数のアーキテクチャ部を切り替える場合の識別子の役割を負っています．しかし，本書ではこの機能は用いないので，すべての識別子をrtlに統一しています．

アーキテクチャ部では，まずsignal文を使って，アーキテクチャ部内で使用する内部信号の宣言を行います．signal文はarchitechtureとそれに続いて記述するbeginのあいだに書きます．**表3-2**の例では，とくに内部信号は使っていないため，notUse0というダミーの信号を宣言してみました．このように，実際に使用していない信号を宣言してもエラーになりませんが，コードが見づらくなるため，本来は好ましくありません．

beginとend rtl; のあいだに書かれているのが，記述の本文です．ここでは，aという信号とbという信号のAND(論理演算)を取り，信号yに反映するという内容の記述を行っています．

RTLレベル
VHDLの記述レベルのひとつ．実際の回路と同様にレジスタの存在を意識して，レジスタ間でデータを処理しながら受け渡していくような記述のこと．おおむね回路合成が可能であると言われている．RTLはRegister Transfer Levelの略なので，本当は末尾に"レベル"をつけるのは間違いか？

signal文
VHDLにおいて，機能モジュール内で使用するローカルな信号(内部信号)の定義に使用される宣言文．

ダミーの信号
ここでは，内部信号の定義を行う位置について説明するため，notUse0というダミーの信号を定義している．本来，この記述では内部信号を使わないので定義を行う必要はない．

VHDLでの信号とは

実際の電子回路では電線や銅箔で信号の伝達を行い，回路図上ではそれらの配線は実線で表されます．CADの回路図上では実線で表示される配線はネットと呼ばれ，それぞれのネットはコンピュータが自動的に付けたネット番号により管理されています．

このように，回路図を用いた設計では，図上で線を引くことにより，配線が存

在することを示すことができました．しかし，回路をテキストで表現するVHDLでは同様の方法を使うことはできません．

そこでVHDLでは，配線の存在を示すために「**配線の名前**」を用います．配線の名前を定義することにより，配線の存在を示すわけです．ちなみに，この配線に相当するもののことをVHDLでは「信号(signal)」と呼びます．信号の名前は，すべて設計者が付けなければなりません．慣れないうちは，この名前付けを煩雑に思われるかもしれません．しかし，回路を記述する過程では，繰り返し信号名を書く必要が生じるため，自分のわかりやすい名前を使うことができることは，むしろ好ましいと言えるのではないでしょうか．

なお，信号名の表記には，アルファベットと数字，そしてアンダースコア(_)を用いることができます．ただし，信号名はかならずアルファベットで始まらなければなりません．また，信号名の末尾にアンダースコア(_)を用いることはできません．

本書の中では，信号名は筆者の好みの記名法を使っています．これは，単語またはそれを略したものを連続して書き(信号名にはスペースを含むことができない)，各単語の先頭の一文字を大文字にしてその区切りを示すというものです．この方法は**マッキントッシュのプログラミングの本**を参考にしました．

しかし，この方法にこだわる必要性はないので，自分でコードを書く場合には，自分の好みの信号名を使ってください．

ネット
net(網)．コンピュータ上の回路図エディタで描いた回路図上の配線は，ネット(回路網の「網」を意味する)と呼ばれる．実際には各ネットは番号により管理され，どの部品とどの部品を接続しているかなどの接続情報なども一緒にファイルに保存される．

配線の名前
VHDLにおいては，回路上のすべての配線(VHDLでは信号と呼ぶ)に設計者が名前を付けなければならない．

マッキントッシュのプログラミングの本
田中太郎著 Think Pascal入門(技術評論社)など(昔の本)．

入出力信号と内部信号とは

入出力信号は，機能モジュールが外界とやり取りをするための信号のことで，モジュール外より機能モジュールにアクセスする場合と，機能モジュール内の記述の両方で使うことができます．

内部信号は機能モジュール内部だけで使われる信号のことです．モジュール外より，この内部信号にアクセスすることはできません．

入出力信号と内部信号は定義する位置が異なります．入出力信号はエンティティ部内において**port文**で定義します．いっぽう，内部信号の定義はアーキテクチャ部の冒頭においてsignal文で行います．

テストなどのために必要な信号は，入出力信号として機能モジュール外より参照できるようにしておくと便利ですが，あまり数が多いと階層間の接続の記述が複雑になります．

port文
VHDLにおいて，機能モジュールの入出力信号を定義するための構文．

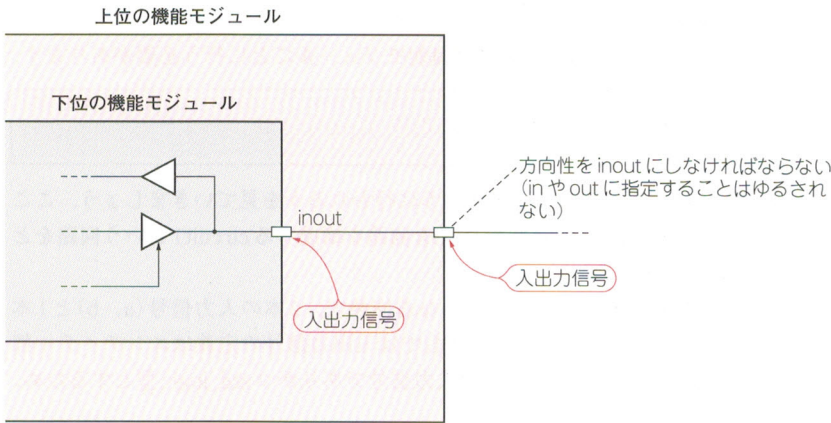
入出力信号の方向性

入出力信号には方向性の指定が必要になります．本書で取り扱う方向性指定はin，out，inoutの3種類です．in，outはそれぞれ入力信号と出力信号を意味し，inoutはデータ・バスなどの双方向性の入出力信号を示しています．

なお，入出力信号の方向性をout(出力信号)とした場合には，機能モジュール内の記述で，この入出力信号への信号代入は可能ですが，出力信号を参照(**信号代入用のソース・データ**としたり，条件判断用に使う)することはできないので注意してください．

また，方向性の指定をinoutとした入出力信号をもつ機能モジュールを，上位の機能モジュールにおいてコンポーネントとして使用する場合(第6章参照)に，下位モジュールのinout指定された入出力信号を，上位のモジュールの入出力信

信号代入用のソース・データ
信号代入文で信号に代入されるデータのこと．信号代入文の右辺に書かれる．

〈図3-4〉 inout指定の連鎖(下位がinoutなら上位もinoutに!!)

号として引き出そうとするときには，その上位モジュールの入出力信号もまた，inout指定にしなければなりません(**図3-4**)．

データ型std_logic(std_logic_vector)の意味

VHDLでは，信号，変数，定数といったデータが使われます．データ型とは，これらのデータがどういう値を取り得るかを示すものです．

コンピュータやディジタル回路では「0か1の世界」と呼ばれるように2値データが用いられます．しかし，実用上，さらにいくつかの状態が表現できないと不便な場合があります．このようなことから，VHDLでは2値およびそれ以外のいくつかの有用な状態を表現できるstd_logic型(およびstd_logic_vector型)が作られました．std_logic型という呼び名は，**ロジック用の標準的なデータ型**という意味から付けられたものでしょう．

std_logic_vector型はベクタ信号用のstd_logic型です．

表3-3にstd_logic型の取り得る状態の一覧を示します．

不思議に思われるかもしれませんが，std_logic型はVHDLの組み込みタイプとしてではなく，**ライブラリ・パッケージ**として用意されています．

このため，std_logic型を使う場合にはかならずつぎの2行の記述により，ライブラリを呼ばなければなりません．

ロジック用の標準的なデータ型
VHDLのデータ型の中でも，実際のディジタル回路の表現に必要なすべての状態を備えたデータ型．VHDLにおけるstd_logic型とstd_logic_vector型のことを指している．

ライブラリ・パッケージ
VHDLが登場した当時は，回路設計用の言語であるにも拘わらず，3ステート('Z')や不定('X')という状態の概念がなかった．その後，必要性が認識されたのか，VHDLの言語仕様にこれらの状態が追加されたが，これらの状態の定義は，VHDL関連のツール本体でなされるのではなく，ライブラリの形で提供されている．

〈表3-3〉 std_logic型で取り扱える状態

状態を表すシンボル	状態名	備考
'U'	初期値	記憶を含む回路(フリップフロップ，レジスタなど)が初期設定されていない状態を示す．'1'であるか'0'であるかわからない
'X'	不定	'1'であるか'0'であるかわからない状態．またはコンフリクトの発生
'0'	'0'レベル	
'1'	'1'レベル	
'Z'	ハイ・インピーダンス	(3ステート状態)
'W'	弱い信号の不定	本書では使用しない
'L'	弱い信号の '0'	
'H'	弱い信号の '1'	
'－'	don't core	

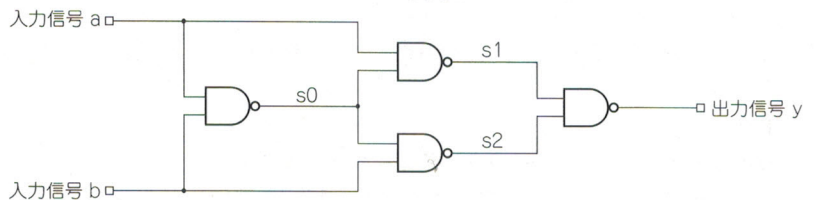
```
library ieee;
use ieee.std_logic_1164.all;
```

なお,このライブラリの使用宣言は機能モジュールごとに行う必要があります.

信号の定義

それでは実際に信号の定義がどのように行われるかを見ていきましょう.ここでは例としてNANDゲート4個でEx-OR回路を構成するcircuit1という回路をとりあげます(**図3-5**).

circuit1は4個のNANDゲートからなる回路で,2本の入力信号(a, b)と1本の出力信号(y)をもっています.VHDLでは入出力信号の定義はエンティティ部内のport文で行います.信号a, bは入力信号でありかつstd_logic型とするため,

〈図3-5〉circuit1の回路とVHDLコード

(a) circuit1の回路図

```
--
-- circuit1
--
library ieee;
use ieee.std_logic_1164.all;

entity circuit1 is
   port(a : in std_logic;
        b : in std_logic;
        y : out std_logic);
end circuit1;

architecture rtl of circuit1 is

   signal s0 : std_logic;
   signal s1 : std_logic;
   signal s2 : std_logic;

begin

   s0 <= a  nand b;
   s1 <= a  nand s0;
   s2 <= b  nand s0;
   y  <= s1 nand s2;

end rtl;
```

(b) circuit1のVHDLコード

```
a : in  std_logic;
b : in  std_logic;
```
信号名 ↑ 信号の型指定
信号の方向性

というような記述になります．出力信号yもstd_logic型にするので，

```
y : out  std_logic
```
信号名 ↑ 信号の型指定
信号の方向性

と記述します．結果として，入出力信号を定義するport文はつぎのようになります．

```
port(a : in  std_logic;
     b : in  std_logic;
     y : out  std_logic);
```

VHDLでは文の末尾は";"(セミコロン)で終わりますが，はじめの2行の行末のセミコロンはそれとは意味が異なり，複数の信号の型指定の間の区切りを表します．この記述は，

```
port(a : in  std_logic;b : in  std_logic;y : out  std_logic);
```
信号の型指定の区切り

という一つの文を，信号の型指定が見やすいように，3行に分けて書いたものなのです．

VHDLにおいては，このように一つの文を数行に分けて書くことがよくあります．その場合，信号の型指定や信号の接続情報の区切りにセミコロン(;)が使われる場合とカンマ(,)が使われる場合があるので注意が必要です．

このあたりのチェックは単体のVHDLシミュレータを使うと，エラーの位置と内容を示してくれるので便利です．回路合成ツールはVHDLの文法よりも，いかに回路に落とすかに主眼をおいて作られているので，文法エラーの指摘に関しては，単体VHDLシミュレータほど親切ではありません．

アーキテクチャ部において，入出力信号以外に信号を使いたい(内部信号)場合には，signal文で内部信号の宣言を行わなければなりません．signal文の書式はつぎのようなものです．

```
signal  s0 : std_logic;
```
信号名 信号の型指定

キーワードのsignalに続けて信号名とその信号の型を指定することになります．

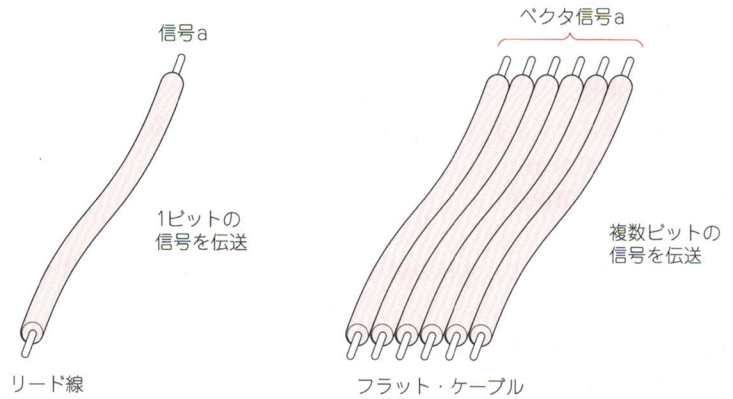
ベクタ信号の扱い方

電子回路の配線は，基本的に1本(1ビット)が単位になっています．しかし，データ・バスやレジスタ，カウンタ，メモリのデータなど，複数ビットのデータをまとめて取り扱うことができると便利な場合があります．

VHDLにおいては，ベクタ信号がこのような要求に応えてくれます．

信号が1本のリード線であるとするなら，ベクタ信号はフラット・ケーブルのようなものだということができます(**図3-6**)．ベクタ信号を使うと複数ビットの信号を一つの信号名で取り扱うことができるので，コードの記述量を少なくする効果があります．ベクタ信号は設計効率を向上するための一つの手段なのです．

ベクタ信号
回路図設計の場合のバスに相当するもの．複数ビットの信号を一つの信号名で取り扱うことができるため，設計効率の向上に役立つ．

〈図3-6〉リード線とフラット・ケーブル

ベクタ信号の型指定(ベクタの範囲指定)

信号が1ビットの信号になるか，ベクタ信号になるかは，信号を定義する際の型指定により決まります．信号の型としてstd_logicを選べば，その信号は1ビットの信号となり，std_logic_vectorを選べば，その信号は複数ビットのベクタ信号となります．

std_logic_vector型を使う場合には，信号の型指定時にベクタの範囲もいっしょに明示しなければなりません．ベクタの範囲は，昇順にすることも降順にすることもできます．また，範囲指定はかならずしも0から始まらなくてもかまいません(**表3-4**)．

2進数の取り扱いの都合からか，電子回路設計においては，一般的にMSB側のビット番号が大きく，LSB側のビット番号が大きい**降順のフォーマット**が使われるようです．降順の場合のベクタ範囲の指定は，

(MSBのビット番号　downto　LSBのビット番号)

という形になります．昇順の場合はdowntoがtoに変わります．

降順のフォーマット
たとえば8ビットの2進数の場合，MSBは2の7乗の重みを，LSBは2の0乗の重みをもつ．このため，8ビットのデータの各ビットにナンバリングを行う場合には，MSBを7，LSBを0とすることが多いようだ．VHDLのビット範囲指定はMSBからLSBへという順番で行われるため，このナンバリングは降順に見えるわけである．

〈表3-4〉std_logic_vector型のフォーマットの例

ベクタのフォーマット	ベクタ信号の型指定	降順/昇順
7 6 5 4 3 2 1 0 (MSB … LSB)	std_logic_vector (7 downto 0)	降順
11 10 9 8 7 (MSB … LSB)	std_logic_vector (11 downto 7)	
0 1 2 3 4 5 6 7 (MSB … LSB)	std_logic_vector (0 to 7)	昇順
3 4 5 (MSB … LSB)	std_logic_vector (3 to 5)	

文字定数

数学における数値に相当するのが，VHDLにおける定数です．定数は基本的には2進数ですが，回路の世界では1，0以外に**3ステート**状態を示すZという値をとることも可能です(Zにはかならず大文字を使うこと)．

文字定数は，1，0またはZの並びをシングル・クォテーション(')またはダブル・クォテーション(")で囲むことにより表します．データが1ビットの場合(std_logic型)にはシングル・クォテーション(')を，データがベクタ(std_logic_vector型)の場合にはダブル・クォテーション(")を用います．

どちらでもよさそうなもののようにも思えますが，VHDLの処理系においては**厳密な取り扱い**が行われているので，きちんと使いわけをしなければなりません(**図3-7**)．

「おいおい，std_logic型ではU(初期値)やX(不定)も使えるはずだ．忘れているよ」なんて言う方はいないでしょうか．U(初期値)は，電源投入直後，イニシャライズされていないフリップフロップやレジスタの出力が1であるか0であるかわからない場合に，シミュレータがそれを知らせるために使う状態です．また，X(不定)は，ゲートの入力を浮かせたとき(Zを与える)や出力同士の**コンフリクト**が起きた場合など，出力や信号の値がどうなるかわからない場合に，シミュレータがそれを知らせるために使う値です．

したがって，実回路の設計において，これらの値(UやX)を使うことはできません．単体VHDLシミュレータでは，これらを含む記述をコンパイルすることができるかもしれません．しかし，回路合成ツールではNGとなります．なぜなら，実回路では実現できない状態だからです．

〈図3-7〉定数の例

"10110100"…2進数の180

"ZZZZZZ"…6ビット全部が3ステート

'1'…………データ'1'

'Z'…………3ステート

3ステート
'1'，'0'の二つの状態に続く第3の状態(ステート)を示す．またの名をハイ・インピーダンス状態とも言い，出力が電源('1'レベル)やグラウンド('0'レベル)から切り離された状況を示す．

厳密な取り扱い
Pascal(プログラム言語の一つ)の流れを汲むVHDLの処理系は文法チェックが厳格であると言われている．Pascalの処理系は，文法チェックが厳しく，間違ったプログラムを組むことが難しいことから，入門者向きの言語と言われていた．しかし，VHDLの場合，文法通りであっても回路合成できるとは限らないため，これがメリットとなっているかどうかには疑問が残る．

コンフリクト
衝突のこと．ディジタル回路においては，'1'の出力と'0'の出力同士をつないでしまい，回路に大電流が流れたりする事態を指す．配線の間違いや3ステート・バスの制御のミスなどによって引き起こされる．

第4章　基本的なゲート回路やこれらの組み合わせ回路の記述法

ロジックの記述…プリミティブな表現

吉澤　清

記憶的な要素
ディジタル回路の中でも，ラッチやフリップフロップ，レジスタなどは，データを記憶する機能をもっている．

コンカレントに動作
実際のディジタル回路は同時並行的(コンカレント)に動作をする．

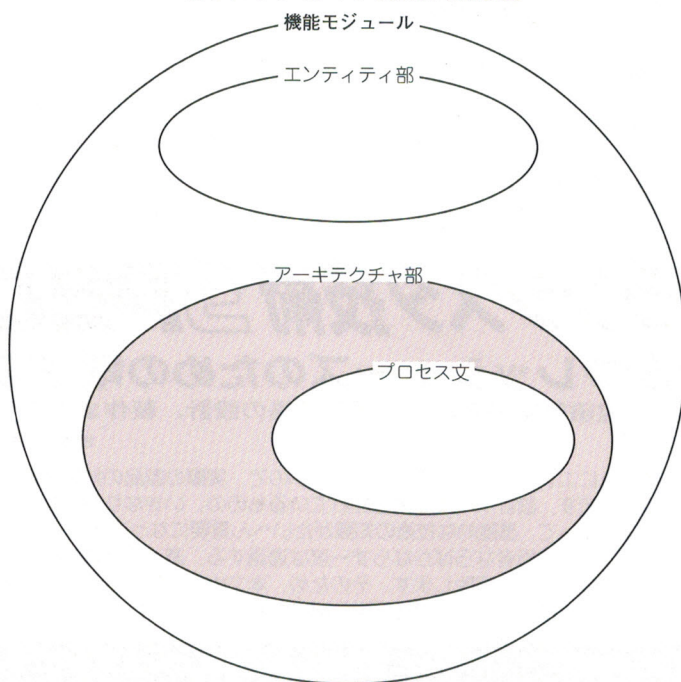
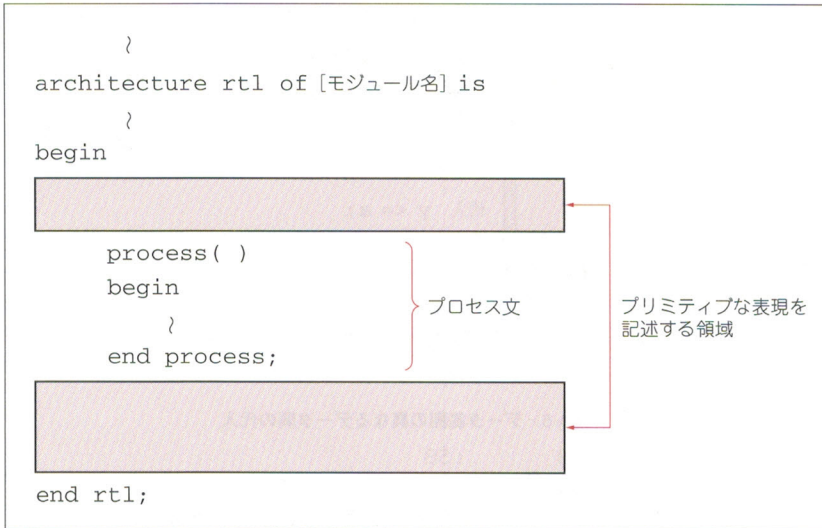
まず，**記憶的な要素**を含まず，入力の値が決まると出力が一義的に決まる回路，つまりゲート回路やそれを組み合わせた回路の記述の仕方について解説します．

本章で述べるのは，プリミティブな表現によるロジックの記述に関してです．プリミティブな表現は，現実の回路に近い記述の仕方であるため，たいがいの回路合成ツールで合成することが可能です．また，回路設計者にとっては，もっともなじみやすい表現であるということができます．

第3章で，VHDLにおいては機能モジュールの中のアーキテクチャ部にモジュールの内部回路の記述を行うと説明しました．アーキテクチャ部には，プロセス文(第5章で解説)により，特別な領域を形成することができますが，ここで説明するプリミティブな表現は，プロセス文以外の領域に記述します(**図4-1**，**図4-2**)．

この領域に記述された内容は，すべて完全に**コンカレント**(同時並行)**に動作**します．したがって，この領域に記述される内容というのは，真にテキストで書かれた回路図ということができます．

VHDL上で使用可能なデータには信号と変数(後述)がありますが，この領域で使うことができるのは信号のみです．変数はシーケンシャルな記述用のデータであり，プロセス文の外で使うことはできません．

〈図4-1〉プリミティブな表現の記述位置

〈図4-2〉プリミティブな表現の記述位置(コード上)

信号代入文の記述の仕方

回路を組み上げていこうとする場合，信号の値や信号の演算結果を別の値へと反映できなければなりません．

たとえば，信号aと信号bのANDを取ったとします．たんにaとbのAND演算を行っただけでは意味がありません．ANDゲートの出力をcならcという信号に接続してはじめて演算結果が生かされるわけです．

VHDLでは，このような「信号への接続」を表現するために，**信号代入文**(<=で表される)を使います．前出の「信号aと信号bのANDをとって，結果を信号cに接続する」という処理をVHDLで書くと，

```
c <= a and b;
```

となります．

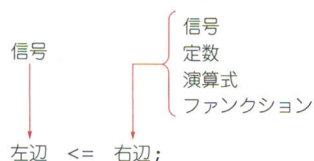
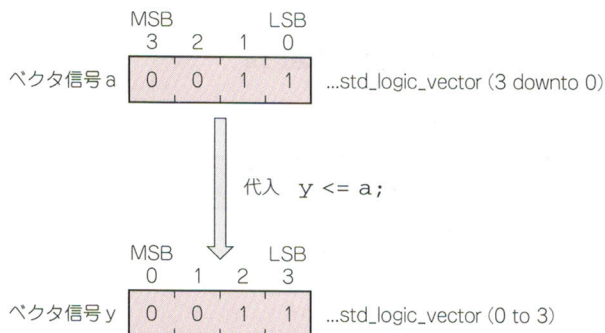
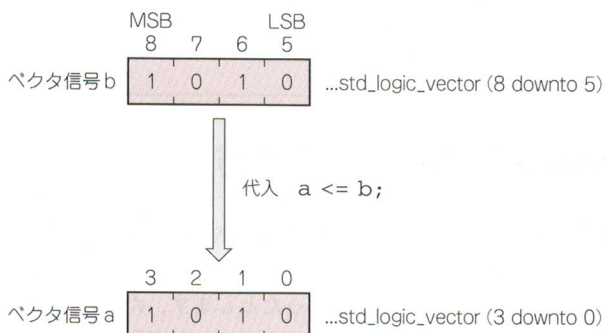
信号代入文の"<="の左辺はかならず信号になります．右辺には信号あるいは定数，演算式，ファンクションなどを記述します(**図4-3**)．"<="の両辺の信号や演算結果などの間では，型が一致していなければならず，また，それがベクタの場合にはベクタ長が一致していなければなりません．型やベクタ長が不一致の場合は，コンパイラではじかれることになります．

ただし，ベクタ信号の降順(downtoを使ったビット範囲指定の場合)と昇順(toを用いたビット範囲指定)に関しては，かならずしも信号代入文の右辺と左辺で一致しなくてもよいようです．

ベクタ信号のビットのナンバリングは，各信号に固有のものであって，そのなかに入るデータについているわけではありません．つまり，ベクタ信号はデータ

信号代入文
VHDLにおいては，配線(信号)の接続の記述は代入の形をとる．

ベクタ信号のビットのナンバリング
ベクタ信号を構成する各ビット・データには，参照をする場合のインデックスとして便宜上ナンバがふられる．

〈図4-3〉信号代入文の書式

〈図4-4〉 昇順，降順の異なるデータ間の代入

〈図4-5〉 データ範囲の異なるデータ間の代入

を入れるための入れ物であり，この入れ物に便宜上番号が付してあるわけです．そして，中に入るデータは単なるビット列ということになります(ただし，std_logic_vector型上では，0や1だけではなく，XやZと言った状態も取りうる)．

したがって，データの昇順，降順が異なるベクタ信号間のデータの代入(**図4-4**)や，データ範囲の異なるベクタ信号間の代入(**図4-5**)も問題なく行うことが可能です．

ベクタ信号にオール1やオール0を代入しようとする場合には，つぎのような表現を使うことができます．

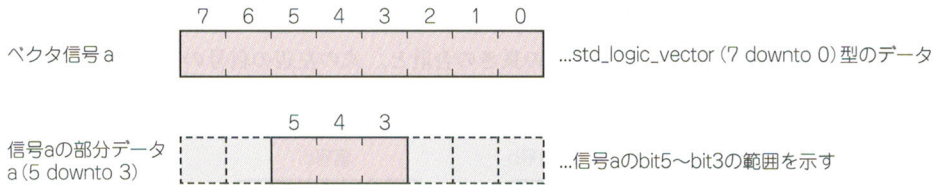
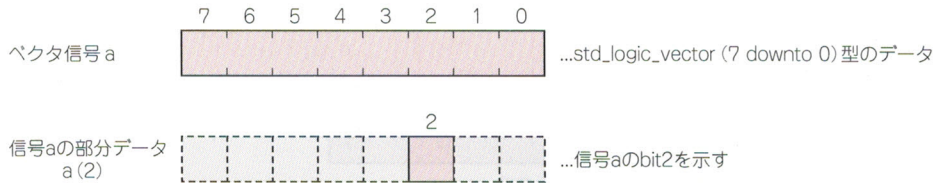
```
a <= (others => '1');        … ベクタ信号aにオール1を代入
b <= (others => '0');        … ベクタ信号bにオール0を代入
```

ここで，a，bは任意のビット幅をもつベクタ信号です．このように，others を使った表現を用いると，ベクタ長を気にすることなくオール1，オール0の代入ができるので，レジスタやカウンタの初期値の設定時などに重宝します．ベクタ長の指定をともなっていないため，ベクタ長を変更する場合などにも，代入文の書き換えが必要となりません．

others
VHDLにおけるキーワードの一つ．「それ以外の場合」を示す．

ベクタ信号の部分データの操作

ベクタ信号の信号名に続けて，ビット範囲指定を書くことにより，ベクタ信号の部分データのみを独立して扱うことができます．たとえば，std_logic_vector (7 downto 0)という型をもったベクタ信号aのbit5からbit3にかけた3ビットのデータを操作しようとする場合には，

〈図4-6〉ベクタ信号とその部分データ

〈図4-7〉ベクタ信号とその特定ビット・データ

```
a(5 downto 3)
```

という形で表現します(**図4-6**).

また，ベクタ信号の中の1ビットのデータのみを扱いたい場合には，信号名に続けてビット位置指定を書きます．ビット位置の指定は，カッコで囲んだビット位置番号です．たとえば，上記のベクタ信号のbit2のみを独立して扱いたい場合には，

```
a(2)
```

と書きます(**図4-7**).

このように表現したベクタ信号の部分データは，通常のデータと同様に扱うことができます．たとえば，

```
a(5 downto 3) <= "110";
a(5 downto 3) <= b;                    (ただし，bは3ビット・データ)
a(2) <= '0';
```

などのように，ベクタ・データの一部分に値を代入することもできますし，また，

```
b <= a(5 downto 3);                    (ただし，bは3ビット・データ)
y <= '1' when (a(5 downto 3) = "000" ) else '0';
                                       (ただし，yは1ビット・データ)
y <= a(2);                             (ただし，yは1ビット・データ)
```

などのように，ベクタ・データの一部分の値を参照することもできます．

当然のことながら，おおもとのベクタ信号のもつビット範囲を超えて，範囲指定を行うことはできません．

連接演算子の使い方

連接演算子(&)を使うと，複数の信号やベクタ信号をつないで，一つのベクタ信号に合成することができます．

連接演算子
複数のデータを繋ぎ合わせるための演算子．

たとえば，

```
a <= "110";            (aは3ビットの信号)
b <= '1';              (bは1ビットの信号)
c <= "1010";           (cは4ビットの信号)
```

```
y <= a & b & c;            (yは8ビットの信号)
```

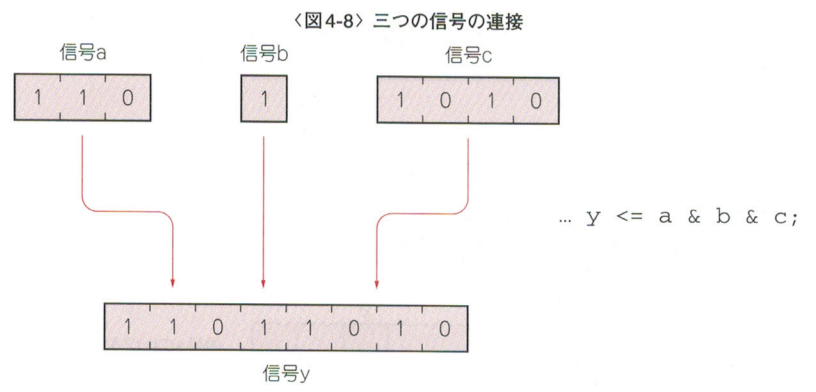
などという記述をすると，yの値は11011010となります(**図4-8**).

なお，式の右辺の信号の長さの合計と，式の左辺の信号の長さが一致していな

〈図4-8〉三つの信号の連接

信号a: 1 1 0　信号b: 1　信号c: 1 0 1 0

… y <= a & b & c;

信号y: 1 1 0 1 1 0 1 0

〈表4-1〉1ビットのシフト/ローテイト

操作	VHDL記述	データの動き
左シフト	`y <= a(2 downto 0) & '0';`	a(3 2 1 0) → y(3 2 1 0) ← '0'
右シフト	`y <= '0' & a(3 downto 1);`	a(3 2 1 0) → '0' → y(3 2 1 0)
左ローテイト	`y <= a(2 downto 0) & a(3);`	a(3 2 1 0) → y(3 2 1 0)
右ローテイト	`y <= a(0) & a(3 downto 1);`	a(3 2 1 0) → y(3 2 1 0)

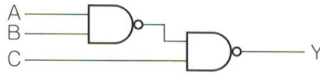
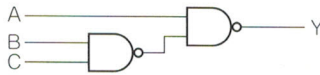
いと，VHDLでは代入ができません．なお，連接しようとする信号のビット範囲指定の降順と昇順が混在しても問題はないようです(downto指定のベクタ信号とto指定のベクタ信号の連接は可能)．

データのシフトとローテイトの記述法

ベクタ信号の部分データの参照と連接演算子を組み合わせることにより，データのシフトやローテイトを行うことができます．たとえば，4ビットのベクタ信号aの値を1ビット・シフトまたはローテイトしてベクタ信号y(当然4ビット)の値とする記述は**表4-1**のようになります．

論理演算子の記述法

VHDLにおいてはnot/and/or/xor/nand/norの6種類の論理演算子を使うことができます(**表4-2**，**表4-3**，**表4-4**)．

演算の優先順位は，not演算がほかの演算にくらべて高く，それ以外の五つの演算は同一レベル(not演算より一段低い)となります．

演算の優先順位
ここでは論理演算の優先順位のことを指す．

〈表4-2〉VHDLの論理演算子一覧

論理演算子	VHDL記述例	相当する論理式	ロジック・シンボル	真理値表
not (否定)	`Y <= not A;`	$Y=\overline{A}$	(NOT回路)	A Y 1 0 0 1
and (論理積)	`Y <= A and B;`	$Y=A\cdot B$	(AND回路)	A B Y 1 1 1 1 0 0 0 1 0 0 0 0
or (論理和)	`Y <= A or B;`	$Y=A+B$	(OR回路)	A B Y 1 1 1 1 0 1 0 1 1 0 0 0
xor (排他的論理和)	`Y <= A xor B;`	$Y=A\oplus B$	(エクスクルーシブOR回路)	A B Y 1 1 0 1 0 1 0 1 1 0 0 0
nand (論理積の否定)	`Y <= A nand B;`	$Y=\overline{A\cdot B}$	(NAND回路)	A B Y 1 1 0 1 0 1 0 1 1 0 0 1
nor (論理和の否定)	`Y <= A nor B;`	$Y=\overline{A+B}$	(NOR回路)	A B Y 1 1 0 1 0 0 0 1 0 0 0 1

注）VHDLの記述例として信号代入文を使用した．現在のVHDLには，“エクスクルーシブNOR”を表す論理演算子はない．

〈表4-3〉誤った記述の例

記述例：Y <= A nand B nand C;	
解説：この記述はコンパイル時のチェックでエラーになる. なぜなら，二つの"nand"演算子の優先順位が等しいため，コンパイラは，この記述が下のA，Bのどちらの回路を表しているのか判断できないからである. VHDLの論理演算においては，一般の数式と異なり，優先順位が同じ演算子が並んでいた場合に，左側から演算を進めていくというような暗黙の了解はない(かっこを使って優先順位を明示する必要がある).	
回路A	回路Aの記述： Y <= (A nand B) nand C;
回路B	回路Bの記述： Y <= A nand (B nand C);
参考：3入力NAND回路は次のように記述する. Y <= not (A and B and C);	

〈表4-4〉まぎらわしい記述の例

記述例：Y <= not A or not B;
解説：この記述は誤りではない．正常に機能する. "not"演算子は"or"演算子よりも優先順位が高いので，この記述は下図のような回路(NAND回路)を示すことになる. しかし，このような書き方は見づらいため，かっこを使って"not"演算の優先度が高いことを明示することをお薦めする. Y <= (not A) or (not B);
上記の記述を示す回路

優先順位の指定は一般的な数式と同じく，カッコ"()"でくくることにより行います．これにより，いちばん内側のカッコでくくられた演算を最優先とすることができます.

論理演算は1ビットのデータのみでなくベクタ・データにも適用することが可能です(たとえば，4ビットのデータ同士のAND演算).

論理ゲートおよび論理ゲートを組み合わせた回路は，論理演算子により記述することができます．また，論理演算子による記述は確実に実回路に合成することができます.

when ～ else文の魔術

when ～ else文は，条件式の判定と**データ・セレクタ**の機能をあわせもつ構文です．

when ～ else文は，**図4-9**に示すような構造をもっています．文の中に書かれた条件式の判定結果により，判定結果がtrueの場合にはデータ1が，判定結果がfalseの場合にはデータ2がwhen ～ else文の結果となります．

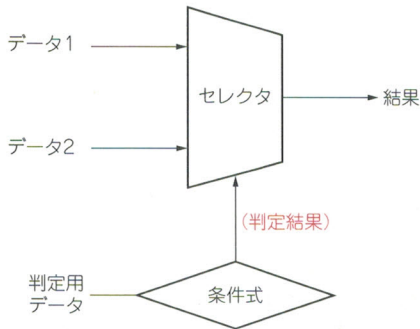
when ～ else文の書式は**図4-10**のようなものです．

ここで，データ1とデータ2は信号または定数です．when ～ else文の選択結果は，信号代入文の形で信号に代入されます．

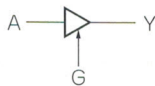
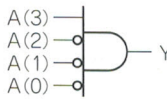
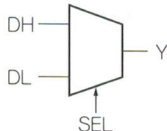
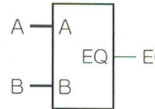
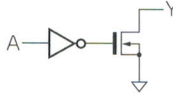
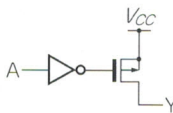
本来，条件式をカッコでくくる必要はありませんが，筆者は自分でコードを見る場合の見やすさを考えて，カッコでくくっています．

見出しに「魔術」という言葉を使いましたが，when ～ else文を憶えるだけで，

データ・セレクタ
制御信号により，複数のデータ入力のうちの1本を選択して出力するような回路のこと．

〈図4-9〉when ～ else文の構造

〈図4-10〉when ～ else文によるデータ選択

```
result <= データ1 when (条件式) else データ2 ;
```
代入先の信号名　trueの場合に選択　falseの場合に選択

条件式の判定結果	選択されるデータ
true	データ1
false	データ2

〈表4-5〉when ～ else文の使い方

機　能	VHDL記述	ロジック・シンボルなど	真理値表など
3ステート・バッファ（1ビット・データの場合）	`Y <= A when (G = '1') else 'Z';`	A, Y, G	G A Y 1 1 1 1 0 0 0 1 Z 0 0 Z
3ステート・バッファ（4ビット・データの場合）	`Y <= A when (G = '1') else "ZZZZ";`		
パターン・マッチ（デコード）	`Y <= '1' when (A = "1000") else '0';`	A(3), A(2), A(1), A(0), Y	Aの値が“1000”の場合にYが‘1’となる
セレクタ	`Y <= DH when (SEL = '1') else DL;`	DH, DL, Y, SEL	SEL DH DL Y 1 1 X 1 1 0 X 0 0 X 1 1 0 X 0 0
一致判定（4ビット・データの場合）	`TEMP <= A xor B;` `EQ  <= '1' when (TEMP = "0000") else '0';`	A, B, EQ	A=Bの場合にEQが‘1’となる
ゲート回路（4ビット・データの場合）	`Y <= A when (G = '1') else "0000";`	A, G, Y	G A Y 1 1 1 1 0 0 0 1 0 0 0 0
N-chオープン・ドレイン・バッファ（1ビット・データの場合）	`Y <= 'Z' when (A = '1') else '0';`	A, Y	A Y 1 Z 0 0
N-chオープン・ドレイン・バッファ（4ビット・データの場合）	`Y <= "ZZZZ" when (A = '1') else "0000";`		
P-chオープン・ドレイン・バッファ（1ビット・データの場合）	`Y <= '1' when (A = '1') else 'Z';`	V_{CC}, A, Y	A Y 1 1 0 Z
P-chオープン・ドレイン・バッファ（4ビット・データの場合）	`Y <= "1111" when (A = '1') else "ZZZZ";`		

注）ここでいうオープン・ドレイン・バッファは，3ステート・バッファを利用したものであり，実際に片側のトランジスタが消えたり，耐圧が上がったりするわけではない．

非常に多様な回路を記述することができます．条件判断とデータ・セレクタの組み合わせは単純ですが，絶妙かつ強力です（**表4-5**）．

多条件のwhen ～ else文の記述法

when ～ else文においては，条件式と信号を複数連続して書くことも可能です．条件を二つもつwhen ～ else文の書式は**図4-11**のようなものです．

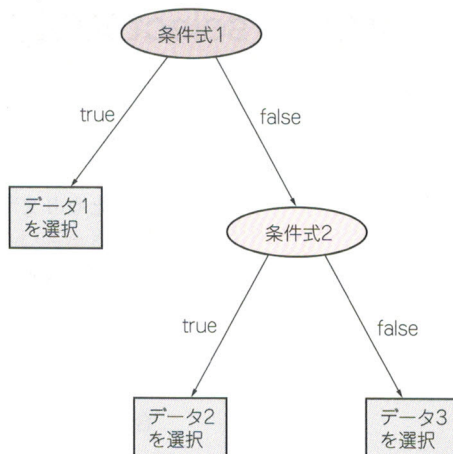
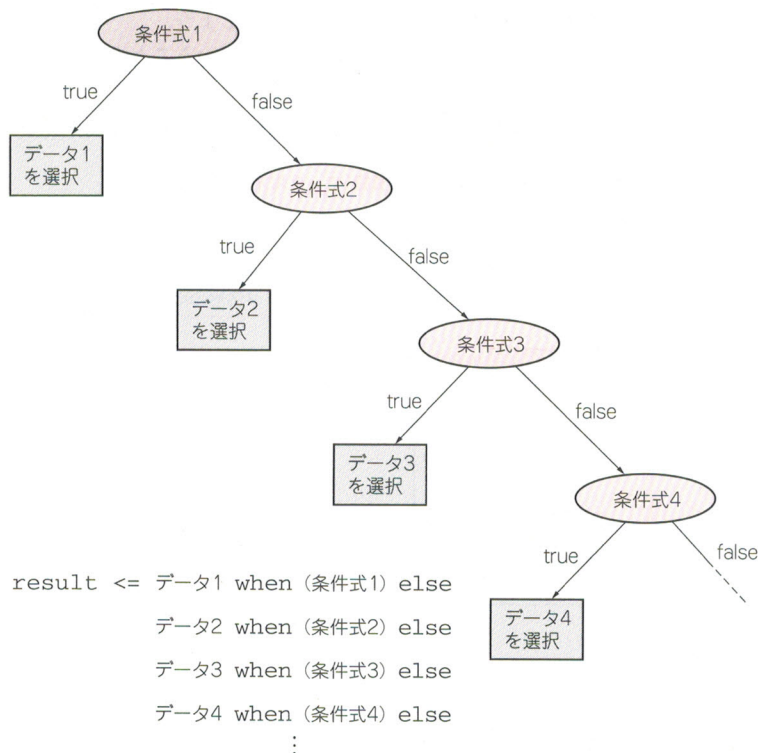
図4-11の書式で書かれたwhen ～ else文の場合，データはつぎのように選択されます（**図4-12**）．条件式は，前のほうから順に判定されていきます．まず，条件式1が評価され，判定結果がtrueの場合にはデータ1が選択されることが確

〈図4-11〉条件を二つもつwhen ～ else文によるデータの選択

```
result <= データ1 when (条件式1) else
          データ2 when (条件式2) else
          データ3 ;
```

（result：代入先の信号名）

条件式の判定結果		選択されるデータ
条件式1	条件式2	
true	―	データ1
false	true	データ2
false	false	データ3

〈図4-12〉条件が二つのwhen ～ else文の働き

〈図4-13〉さらに条件が増えると…

定します．判定結果がfalseの場合には，つぎに書かれた条件式(条件式2)の評価が行われます．そして，条件式1がfalseで条件式2がtrueの場合にはデータ2が選択され，条件式1がfalseで条件式2もfalseの場合には，最後に書かれたデータ3が選択されます．

データと条件式はさらに続けて書くことも可能です(**図4-13**)．多条件のwhen ～ else文は，多入力のデータ・セレクタやプライオリティ・エンコーダなどの記述に用いられます．

プライオリティ・エンコーダ
エンコーダは符号化器の意．通常，入力に優先順位がついていることが多い．本章の後半に記述例あり．

with ～ select文の記述法

キーワード
VHDLにおける予約語(何らかの意味や機能をもつ単語)のこと.

信号代入文の一種
when～else文やwith～select文は，特定のデータにより複数のデータのうちの一つを選択する機能をもっている．選択されたデータはかならず信号に代入されるため，when～else文やwith～select文も信号代入文の一種と言うことができる.

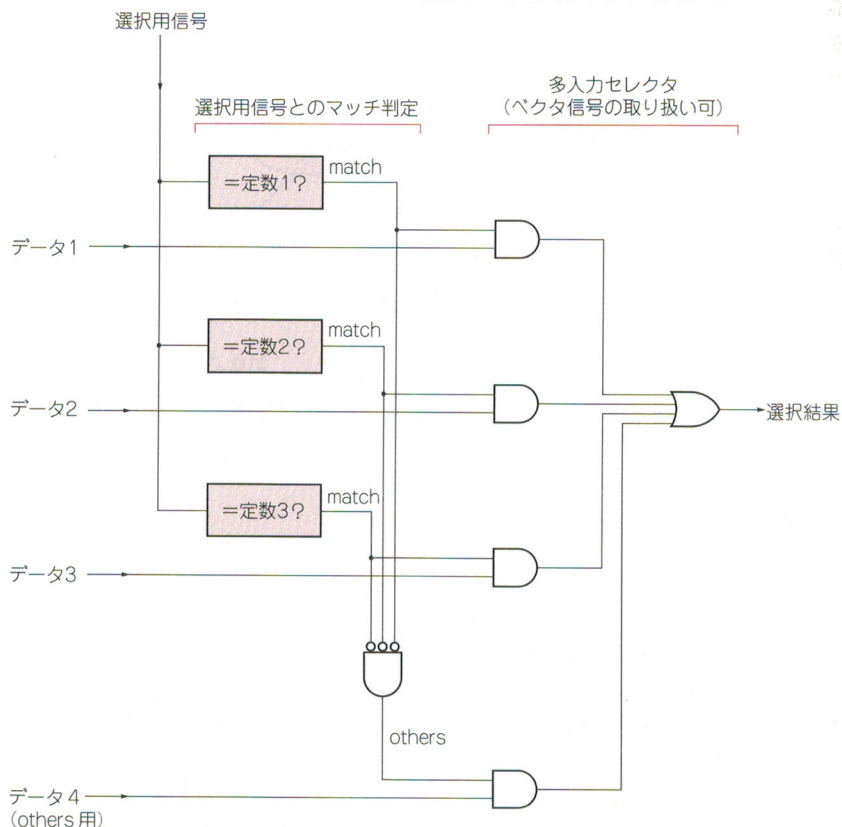
with ～ select文は，特定の信号の値により，複数のデータの中の一つを選択するための構文です．被選択データは定数とペアで記述され，特定の信号(選択用信号)と定数が一致した場合に，定数とペアを組んでいたデータが選択され，結果となります．構文の末尾に書かれるデータに関しては，定数ではなくothersという**キーワード**と組みにして書かれます．othersは文字通り「それ以外」の意味をもち，選択用信号がそれまでに書かれていた定数のいずれともマッチしなかった場合を指します．この場合には，末尾のデータが選択されることになります(**図4-14**).

with ～ select文の書式は**図4-15**のようなものです(マッチ用定数が3個の場合).

withとselectの間に書かれた選択用信号の値が，whenの右側に書かれた定数とマッチした場合，whenを隔てて定数の左側に書かれたデータが選択され，結果として代入されます．with ～ select文は，構文の中に代入先の信号名の記述があるので，**信号代入文の一種**ということができるでしょう.

被選択用データとしては，信号と定数のいづれかを用いることができます.

データが信号である場合には，with ～ select文は多入力セレクタとして働きます．データが定数の場合は，with ～ select文はデコーダとして働いたり，ま

〈図4-14〉 with ～ select文の構造(マッチ用定数が3個の場合)

備考）被選択用データとしては，信号または定数が使用可能

〈図4-15〉with ～ select文によるデータ選択

```
with 選択用信号名 select
    result <= データ1 when 定数1,
              データ2 when 定数2,
              データ3 when 定数3,
              データ4 when others;
```

代入先の信号名　　データと定数がペアとなる

選択用信号の値	選択結果
=定数1	データ1
=定数2	データ2
=定数3	データ3
上記以外の場合	データ4

た，真理値表に基づいたロジック回路の記述に用いられたりします．

真理値表は既知であるけれども，その機能を実現するためにどんな回路を組まなければならないかわからないという場合，with ～ select文を用いると，真理値表を基にロジック回路の機能記述を行うことができます．

この種の記述をながめると，まるで**テーブルROM**が合成されてしまいそうにも思えますが，最近の回路合成ツールの論理圧縮能力は凡人(たとえば筆者)のそれを凌ぐレベルに達しているため，可能であればもっと単純な回路に落としてくれます．

テーブルROM
ROM(Read Only Memory)を数値の表(テーブル)として使ったもの．

もっとも，CPLDの場合，ロジック回路の機能を実現してくれるのはPLAであるため，結局テーブルROM的な記述とのギャップのほうが少ないのかもしれません．

ともあれ，このような手法を使っても，これまでは設計できなかったものが，設計できるようになるのであれば，それはそれでVHDLのご利益と考えることができるのではないでしょうか．

真理値表を基に機能モジュールを記述する

それでは具体的に真理値表のデータを基にVHDLコードを書いてみましょう．ここでは例として，演算回路の基本となるフルアダー(**全加算器**)回路を取り上げます．フルアダーのシンボルと真理値表は**図4-16**の通りです．

全加算器
桁上げ信号(キャリ)の伝搬を伴う2数の加算を行うのに必要な機能をもった加算器．

フルアダーは3本の入力と2本の出力をもつ回路です．その動作は，図の真理値表の通りです．たとえば，B入力＝‘1’，A入力＝‘0’，CI入力＝‘1’の場合，出力はCO＝‘1’，S＝‘0’となります．

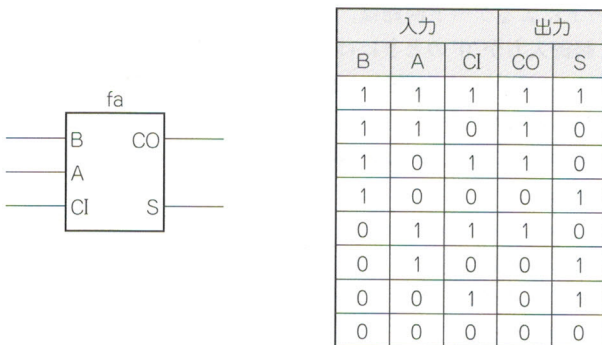
入力が3本であることから，入力データの組み合わせは8通りとなります(入力

〈図4-16〉フルアダー

(a) 記号

入力			出力	
B	A	CI	CO	S
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

(b) 真理値表

〈リスト4-1〉真理値表をもとにしたフルアダー回路

```
--
-- full adder from truth table
--
library ieee;
use ieee.std_logic_1164.all;

entity faT is
          port(   a   : in std_logic;
                  b   : in std_logic;
                  ci  : in std_logic;
                  co  : out std_logic;
                  s   : out std_logic);
end faT;

architecture rtl of faT is

          signal tableIn   : std_logic_vector(2 downto 0);
          signal tableOut  : std_logic_vector(1 downto 0);

begin

          tableIn <= b & a & ci;

          with tableIn select
                  tableOut <=         "11" when "111",
                                      "10" when "110",
                                      "10" when "101",
                                      "01" when "100",
                                      "10" when "011",
                                      "01" when "010",
                                      "01" when "001",
                                      "00" when others;

          co <= tableOut(1);
          s  <= tableOut(0);

end rtl;
```

が'1'か'0'の2値データの場合)．この8通りの入力と出力の関係をwith ～ select文で記述することができれば実際に働くフルアダーを得ることができます(**リスト4-1**)．

ゲート回路の記述法

ゲート回路
Gateは門の意．データを伝達したり，マスクしたりする回路のこと．

ANDゲートやORゲートを使うと，これらの回路の呼び名の由来である**ゲート回路**を実現することができます．ゲート回路とは，制御信号により入力信号を出力に伝達したり，しなかったりするような回路のことです．

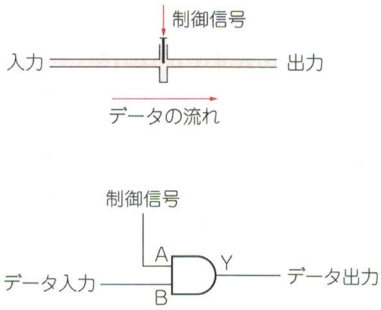
ANDゲートによるゲート回路は，制御入力が'1'のときに入力データが出力に伝えられ，制御入力が'0'のときには出力が'0'に固定されます(**図4-17**)．

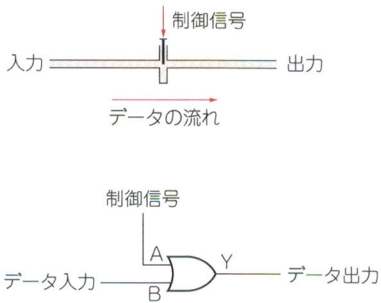
ORゲートによるゲート回路は，制御入力が'0'のときに入力データが出力に伝えられ，制御入力が'1'のときには出力が'1'に固定されます(**図4-18**)．

ゲート回路はANDまたはORの論理演算なので，取り扱うデータが1ビットの信号の場合には，

```
dOut <= dIn and control;
```

(dIn　　：入力信号
control：ゲート制御信号
dOut　：出力信号
すべての信号はstd_loghic型)

〈図4-17〉ANDゲートによるゲート回路

制御信号	データ入力	データ出力	動作
A	B	Y	
1	1	1	入力データを出力に伝達
1	0	0	
0	1	0	出力は'0'に固定
0	0	0	

〈図4-18〉ORゲートによるゲート回路

制御信号	データ入力	データ出力	動作
A	B	Y	
1	1	1	出力は'1'に固定
1	0	1	
0	1	1	入力データを出力に伝達
0	0	0	

というように書くことができます．dInとcontrolのANDをとってdOutへ返すわけです．また，取り扱うデータがベクタ信号の場合は，when ～ else文を使って，たとえば，

```
dOut <= dIn when (control = '1') else "0000";
```

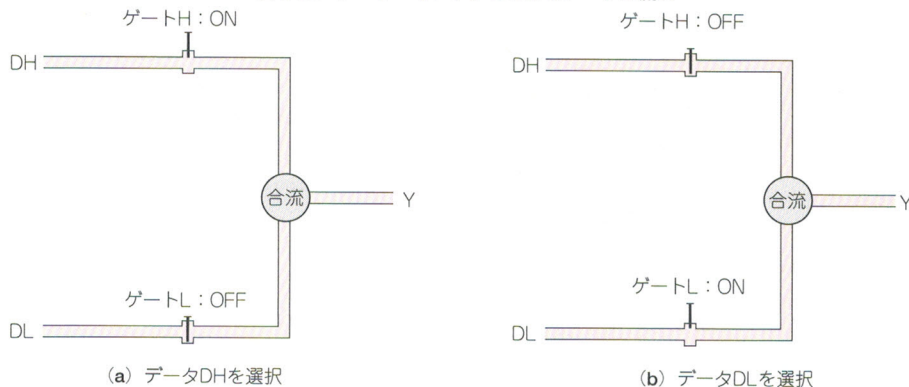
（dIn　　：入力信号（4ビットのstd_logic_vector型）
control：ゲート制御信号（std_logic型）
dOut　：出力信号（4ビットのstd_logic_vector型））

というように書くことができます．

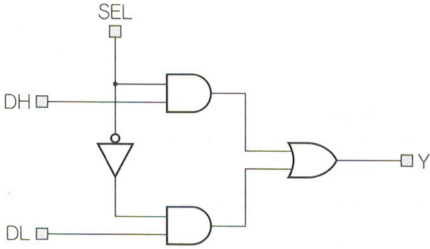
データ・セレクタ（マルチプレクサ）の記述法

二つのゲート回路を用意し，いっぽうには制御信号をそのまま，そしてもう一方には制御信号を反転して与えると，片方のゲートが開いているときにはもう片

〈図4-19〉データ・セレクタにおけるデータの流れ

(a) データDHを選択　(b) データDLを選択

方のゲートは閉じているといった回路が作れます．双方のゲートの出力データをうまく合流させることができれば，制御信号により二つの入力データのいずれかを選択することができる回路，データ・セレクタが得られます(**図4-19**)．

ゲート回路にANDを使う場合，ゲートを閉じたときの出力は‘0’になるため，どちらかの入力が‘1’になった場合に出力が‘1’となるORゲートで，二つのゲートの出力を合流させることができます．データ・セレクタの回路と真理値表を図

〈図4-20〉データ・セレクタ

(**a**) 回路図

制御信号	データ入力		データ出力	動作
SEL	DH	DL	Y	
1	1	×	1	DHを選択
1	0	×	0	
0	×	1	1	DLを選択
0	×	0	0	

(**b**) 真理値表

〈リスト4-2〉データ・セレクタ

```
--
-- 2-In data selector
--
library ieee;
use ieee.std_logic_1164.all;

entity sel2 is
          port(   dh  : in std_logic;
                  dl  : in std_logic;
                  sel : in std_logic;
                  y   : out std_logic);
end sel2;

architecture rtl of sel2 is
begin

          y <= dh when (sel = '1') else dl;

end rtl;
```

〈リスト4-3〉データ・セレクタ(ベクタ信号の選択)

```
--
-- 2-In data selector (select 4-bit vector signal)
--
library ieee;
use ieee.std_logic_1164.all;

entity sel2_vec is
          port(   dh  : in std_logic_vector(3 downto 0);
                  dl  : in std_logic_vector(3 downto 0);
                  sel : in std_logic;
                  y   : out std_logic_vector(3 downto 0));
end sel2_vec;

architecture rtl of sel2_vec is
begin

          y <= dh when (sel = '1') else dl;

end rtl;
```

4-20に示します．

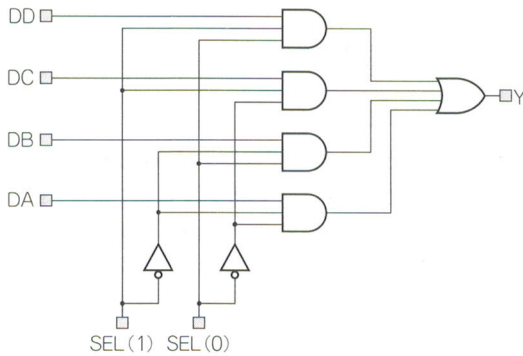
ここでは，図や表の作成の都合上，被選択データが1ビットの信号の場合について示していますが，ベクタ信号の選択を行うことも可能です(**リスト4-2**，**リスト4-3**)．

多入力のデータ・セレクタの記述法

たとえば，2ビットのデータにより4本の入力のうちの1本を選択して出力する**4-to-1データ・セレクタ**は，**図4-21**のような回路と真理値表をもちます．

バイナリ・データにより，複数の入力データのうちの一つを選択する場合，回路の構成はデコーダとデータ・セレクタをいっしょにしたようなものとなります．2入力のデータ・セレクタの場合と同様に，ベクタ信号の取り扱いも可能です(**リスト4-4**，**リスト4-5**)．

4-to-1データ・セレクタ
制御信号により，四つの入力データのうちの一つを選択して出力する回路のこと．

〈図4-21〉4-to-1データ・セレクタ

(a) 回路図

制御信号		データ入力				データ出力	動作
SEL(1)	SEL(0)	DD	DC	DB	DA	Y	
1	1	1	×	×	×	1	DDを選択
1	1	0	×	×	×	0	
1	0	×	1	×	×	1	DCを選択
1	0	×	0	×	×	0	
0	1	×	×	1	×	1	DBを選択
0	1	×	×	0	×	0	
0	0	×	×	×	1	1	DAを選択
0	0	×	×	×	0	0	

(b) 真理値表

〈リスト4-4〉4-to-1データ・セレクタ(with ～ select文による記述)

```
--
-- 4-to-1 data selector (use with-select statement)
--
library ieee;
use ieee.std_logic_1164.all;

entity sel4_A is
          port(   dD  : in std_logic;
                  dC  : in std_logic;
                  dB  : in std_logic;
                  dA  : in std_logic;
                  sel : in std_logic_vector(1 downto 0);
                  y   : out std_logic);
end sel4_A;

architecture rtl of sel4_A is
begin

          with sel select
                  y <=     dD when "11",
                           dC when "10",
                           dB when "01",
                           dA when others;

end rtl;
```

〈リスト4-5〉4-to-1データ・セレクタ(多条件when ～ else文による記述)

```
--
-- 4-to-1 data selector (use when-else statement)
--
library ieee;
use ieee.std_logic_1164.all;

entity sel4_B is
          port(   dD  : in std_logic;
                  dC  : in std_logic;
                  dB  : in std_logic;
                  dA  : in std_logic;
                  sel : in std_logic_vector(1 downto 0);
                  y   : out std_logic);
end sel4_B;

architecture rtl of sel4_B is
begin

          y <=    dD when (sel = "11") else
                  dC when (sel = "10") else
                  dB when (sel = "01") else
                  dA;

end rtl;
```

加算器(フルアダー)の記述法

加算器(フルアダー)は演算回路の基本となる構成要素です(**図4-22**).

AとBの2本のデータ入力，そして下位ビットよりの桁上げ入力であるCI，この三つの値を足して加算結果である出力Sと，上位ビットへの桁上げ用データCOを出力するのがフルアダー回路の働きです(**図4-23**).

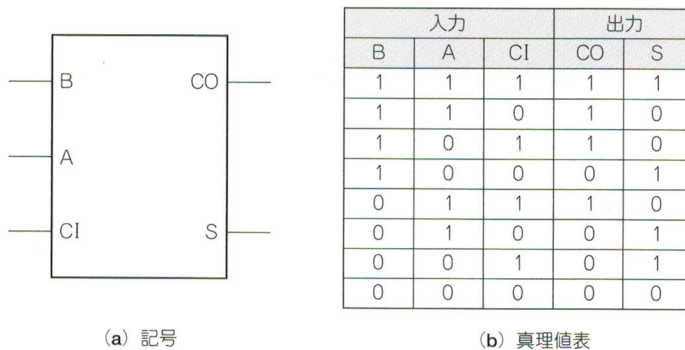
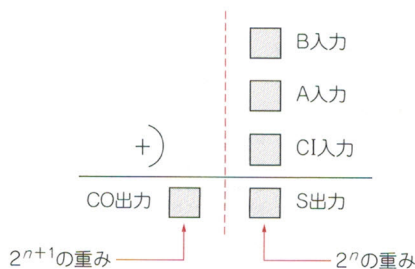
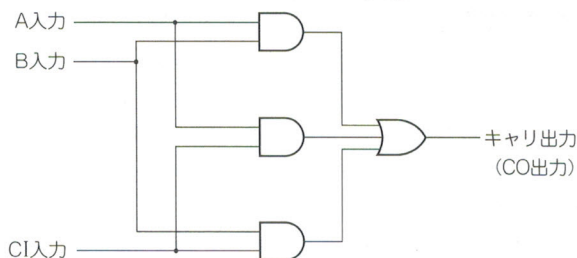
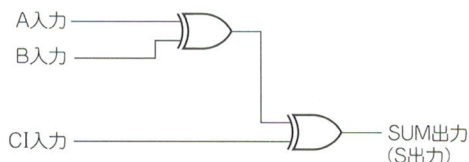
機能的には，各入力の値を足し上げていって，2以上となった場合には**キャリ**

キャリ出力
加算回路において，上位ビットに対する桁上げ信号のことをキャリと呼ぶ.

〈図4-22〉加算器(フルアダー)

(a) 記号

入力			出力	
B	A	CI	CO	S
1	1	1	1	1
1	1	0	1	0
1	0	1	1	0
1	0	0	0	1
0	1	1	1	0
0	1	0	0	1
0	0	1	0	1
0	0	0	0	0

(b) 真理値表

〈図4-23〉フルアダー回路

〈図4-24〉キャリ生成回路

〈図4-25〉SUM出力生成回路

出力を出し，余りをS出力とするわけです．したがって，キャリ出力はAとB，あるいはAとCIまたはBとCIが同時に'1'となっている場合に'1'とすればよいわけです．この通り，そのままを回路にすると**図4-24**のようになります．

S出力に関しては，各入力の足し上げ結果が0，1の場合にはこれらの値がそのまま出力されます．各入力の足し上げ結果が2，3の場合には，上位ビットへの桁上げが発生するため，S出力へは足し上げ結果より2を引いた値，それぞれ0，1が出力されます．結局，S出力が'1'となるのは，各入力の足し上げ結果が奇数になる場合であるということができます．

3本の入力をもち，'1'が入力されている本数が奇数になった場合に出力が'1'となるような回路は，**図4-25**のようにEx-ORを**シリーズに接続**することにより得ることができます．

シリーズに接続
縦続接続の意．

ハーフアダーと区別するために，フルアダーのことを全加算器(直訳か？)と呼ぶことがあります(**リスト4-6**)．

〈リスト4-6〉フルアダー

```
--
-- full adder
--
library ieee;
use ieee.std_logic_1164.all;

entity fa is
          port(   a   : in std_logic;
                  b   : in std_logic;
                  ci  : in std_logic;
                  co  : out std_logic;
                  s   : out std_logic);
end fa;

architecture rtl of fa is
begin

          co <= (a and b) or ( (a or b) and ci);
          s  <= a xor b xor ci;

end rtl;
```

加算器(ハーフアダー)の記述法

半加算器

全加算器(フル・アダー)は半加算器(ハーフ・アダー)2個により構成することができる．このことから半加算器の名がある．

ハーフアダーはフルアダーにくらべ入力の本数が1本少ない加算器です．回路の構成から**半加算器**と呼ばれることがあります(**図4-26**)．

ハーフアダーは下位よりのキャリの伝搬がない場合や，同一の重みのデータ入力が1本しかない場合に使われます．

ここでは，入力信号名をA入力とB入力としますが，加算器の入力の機能はすべて同じなので(キャリの伝搬の都合上，キャリの入出力間の伝搬遅延が少なくなるように設計される傾向はある)，桁上げ入力が必要な場合には，B入力の名称をCI入力と変更してもなんら差し支えありません(**図4-27**，**リスト4-7**)．

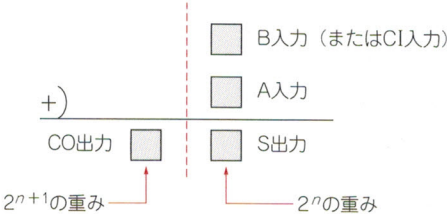
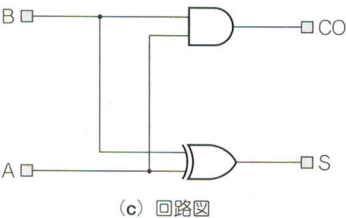
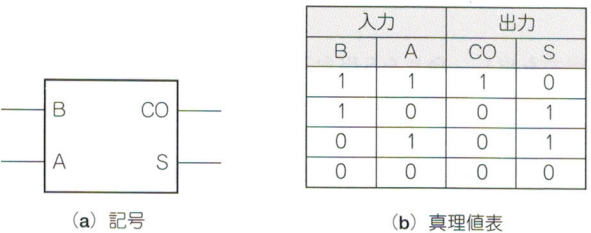
〈図4-26〉加算器(ハーフアダー)

〈図4-27〉ハーフアダー回路

(a) 記号

入力		出力	
B	A	CO	S
1	1	1	0
1	0	0	1
0	1	0	1
0	0	0	0

(b) 真理値表

(c) 回路図

〈リスト4-7〉ハーフアダー

```
--
-- half adder
--
library ieee;
use ieee.std_logic_1164.all;

entity ha is
          port(    a   : in std_logic;
                   b   : in std_logic;
                   co  : out std_logic;
                   s   : out std_logic);
end ha;

architecture rtl of ha is
begin

          co <= a and b;
          s  <= a xor b;

end rtl;
```

デコーダの記述法

入力された2進数が特定の値であった場合に出力を**アクティブ**にし，それ以外の値であった場合に出力を**非アクティブ**とする．そのような操作をデコードと呼びます．たとえば，3ビットの入力データが"110"であったときに，出力が1に，それ以外の値であった場合には出力が0となる回路は，**図4-28**のようなものとなります．

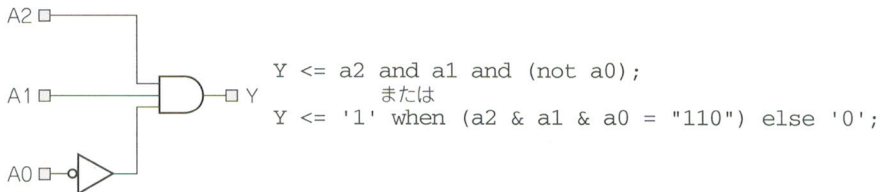
この回路においてY出力はA2 = 1，A1 = 1，A0 = 0の場合にのみ1となり，

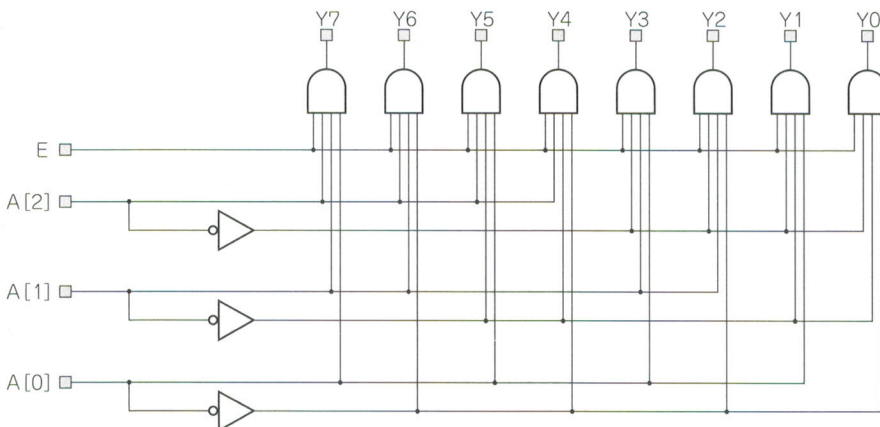
アクティブ
正論理の場合には‘1’，負論理の場合には‘0’のこと．

非アクティブ
正論理の場合には‘0’，負論理の場合には‘1’のこと．

〈図4-28〉"110"のデコード

〈図4-29〉3-to-8デコーダ

(a) 記号

入力				出力							
E	A2	A1	A0	Y7	Y6	Y5	Y4	Y3	Y2	Y1	Y0
1	1	1	1	1	0	0	0	0	0	0	0
1	1	1	0	0	1	0	0	0	0	0	0
1	1	0	1	0	0	1	0	0	0	0	0
1	1	0	0	0	0	0	1	0	0	0	0
1	0	1	1	0	0	0	0	1	0	0	0
1	0	1	0	0	0	0	0	0	1	0	0
1	0	0	1	0	0	0	0	0	0	1	0
1	0	0	0	0	0	0	0	0	0	0	1
0	×	×	×	0	0	0	0	0	0	0	0

(b) 真理値表

(c) 回路図

それ以外の場合には0となります.

このような回路も広義の意味のデコーダになるわけですが，ディジタル回路においてデコーダというときは，一般的にフルデコードをする回路をさすことが多いようです．これは，標準ロジックICの影響によるところが多いのではないかと思います.

標準ロジックICが世に出た当時は，回路の書き換えなどというのは夢の話でしたので，汎用性をもったデコード回路としては，入力のすべての組み合わせに対してデコード出力をもたせたフルデコード回路とせざるを得なかったものと思われます.

それでは，フルデコード回路について見ていきましょう．たとえば，入力データが3ビットの**イネーブル付き**のフルデコード回路は，8本(2^3本)の出力をもちます(**図4-29**).

VHDLを使って設計を行う場合には，フルデコーダを記述してもよいでしょうし，使わない出力は省略したり，デコード条件に冗長性をもたせてもよいでしょう.

フルデコーダを記述した場合でも，使用していない出力については，回路合成ツールが論理圧縮の段階で回路を削除してくれます(**リスト4-8**，**リスト4-9**，**リスト4-10**).

イネーブル付き

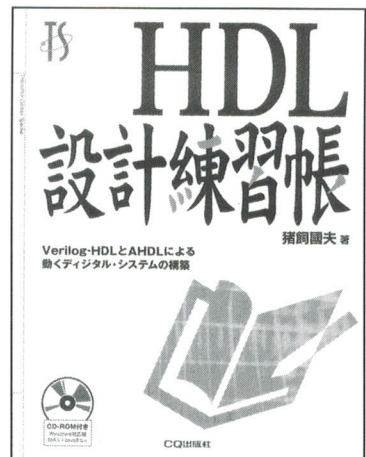
出力イネーブルのこと．イネーブル入力が‘1’の場合にのみ，選択された出力がアクティブとなり，イネーブル入力が‘0’の場合には全出力が非アクティブとなる.

〈リスト4-8〉3-to-8デコーダ(論理演算子による記述)

```
--
-- 3-to-8 decoder (use logical equation)
--
library ieee;
use ieee.std_logic_1164.all;

entity dec3_A is
          port(   a  : in std_logic_vector(2 downto 0);
                  e  : in std_logic;
                  y7 : out std_logic;
                  y6 : out std_logic;
                  y5 : out std_logic;
                  y4 : out std_logic;
                  y3 : out std_logic;
                  y2 : out std_logic;
                  y1 : out std_logic;
                  y0 : out std_logic);
end dec3_A;

architecture rtl of dec3_A is

begin

          y7 <= e and      a(2)  and      a(1)  and      a(0);
          y6 <= e and      a(2)  and      a(1)  and (not a(0));
          y5 <= e and      a(2)  and (not a(1)) and      a(0);
          y4 <= e and      a(2)  and (not a(1)) and (not a(0));
          y3 <= e and (not a(2)) and      a(1)  and      a(0);
          y2 <= e and (not a(2)) and      a(1)  and (not a(0));
          y1 <= e and (not a(2)) and (not a(1)) and      a(0);
          y0 <= e and (not a(2)) and (not a(1)) and (not a(0));

end rtl;
```

〈リスト4-9〉3-to-8デコーダ(when ～ else文による記述)

```
--
-- 3-to-8 decoder (use when-else statement)
--
library ieee;
use ieee.std_logic_1164.all;

entity dec3_B is
          port(   a   : in std_logic_vector(2 downto 0);
                  e   : in std_logic;
                  y7  : out std_logic;
                  y6  : out std_logic;
                  y5  : out std_logic;
                  y4  : out std_logic;
                  y3  : out std_logic;
                  y2  : out std_logic;
                  y1  : out std_logic;
                  y0  : out std_logic);
end dec3_B;

architecture rtl of dec3_B is

          signal inBuf   : std_logic_vector(3 downto 0);

begin

          inBuf <= e & a;

          y7 <= '1' when (inBuf = "1111") else '0';
          y6 <= '1' when (inBuf = "1110") else '0';
          y5 <= '1' when (inBuf = "1101") else '0';
          y4 <= '1' when (inBuf = "1100") else '0';
          y3 <= '1' when (inBuf = "1011") else '0';
          y2 <= '1' when (inBuf = "1010") else '0';
          y1 <= '1' when (inBuf = "1001") else '0';
          y0 <= '1' when (inBuf = "1000") else '0';

end rtl;
```

〈リスト4-10〉3-to-8デコーダ(with ～ select文による記述)

```
--
-- 3-to-8 decoder (use with-select statement)
--
library ieee;
use ieee.std_logic_1164.all;

entity dec3_C is
          port(    a   : in std_logic_vector(2 downto 0);
                   e   : in std_logic;
                   y7  : out std_logic;
                   y6  : out std_logic;
                   y5  : out std_logic;
                   y4  : out std_logic;
                   y3  : out std_logic;
                   y2  : out std_logic;
                   y1  : out std_logic;
                   y0  : out std_logic);
end dec3_C;

architecture rtl of dec3_C is

          signal inBuf   : std_logic_vector(3 downto 0);
          signal outBuf  : std_logic_vector(7 downto 0);

begin

          inBuf <= e & a;

          with inBuf select
                 outBuf <=           "10000000" when "1111",
                                     "01000000" when "1110",
                                     "00100000" when "1101",
                                     "00010000" when "1100",
                                     "00001000" when "1011",
                                     "00000100" when "1010",
                                     "00000010" when "1001",
                                     "00000001" when "1000",
                                     "00000000" when others;

          y7  <= outBuf(7);
          y6  <= outBuf(6);
          y5  <= outBuf(5);
          y4  <= outBuf(4);
          y3  <= outBuf(3);
          y2  <= outBuf(2);
          y1  <= outBuf(1);
          y0  <= outBuf(0);

end rtl;
```

7セグメント・デコーダの記述法

近年では，**ドット・マトリクス・ディスプレイ**も広まりつつありますが，7セグメント・タイプのLEDやLCDは，ドライブが容易であることもあり，表示器としてはまだまだ主流であると言えます．

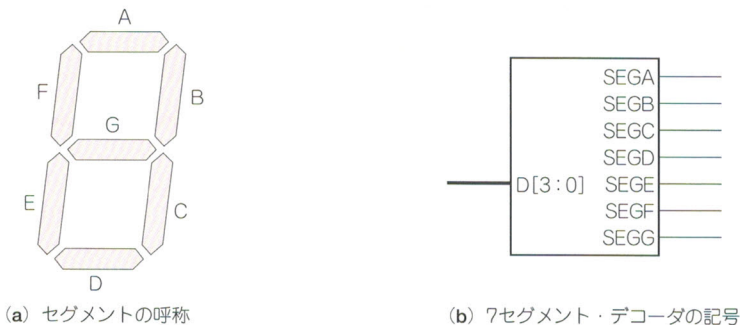
2進データが与えられると，そのデータに対応した数字を表示するためのセグメント・パターンを出力するのが7セグメント・デコーダです(**図4-30，リスト4-11**)．

発光ダイオードは本来2端子のデバイスであり，7セグメントLED表示器の場合は1桁あたり7～8個の発光ダイオードが集積されています．各セグメントあたり2本の端子を引き出すのは，配線を行う点からも煩雑なため，一般的な7セグメントLED表示器では発光ダイオードの**アノード**または**カソード**のいづれか

ドット・マトリクス・ディスプレイ
点の集合により，文字を表示するタイプの表示器．7セグメント表示器にくらべ，表示可能な文字種が多いが，その制御は複雑となる．

アノード
Anode．陽極．電子デバイスにおいてプラスの電圧を印加する電極のことを指す．

〈図4-30〉7セグメント・デコーダ

(a) セグメントの呼称　　(b) 7セグメント・デコーダの記号

入力				出力							
D[3]	D[2]	D[1]	D[0]	SEGA	SEGB	SEGC	SEGD	SEGE	SEGF	SEGG	表示字体
0	0	0	0	1	1	1	1	1	1	0	0
0	0	0	1	0	1	1	0	0	0	0	1
0	0	1	0	1	1	0	1	1	0	1	2
0	0	1	1	1	1	1	1	0	0	1	3
0	1	0	0	0	1	1	0	0	1	1	4
0	1	0	1	1	0	1	1	0	1	1	5
0	1	1	0	1	0	1	1	1	1	1	6
0	1	1	1	1	1	1	0	0	0	0	7
1	0	0	0	1	1	1	1	1	1	1	8
1	0	0	1	1	1	1	1	0	1	1	9
上記以外				0	0	0	0	0	0	0	(ブランク)

(c) 7セグメント・デコーダ(カソード・コモンLED用)の真理値表

が共通となっています．カソード側を共通の端子としたものをカソード・コモン型，アノード側を共通としたものをアノード・コモン型と呼びます．

今回取り上げたデコーダはカソード・コモンLED用なので，**電流制限用抵抗**を介してカソード・コモン型LEDに接続することにより，表示器の駆動が可能です(**図4-31**)．

発光ダイオードの順方向電圧V_fは従来からある赤や緑のものの場合2V台の値であるため，5V系の出力バッファでそのまま駆動しようとすると，過大な電流が流れ，表示器や駆動側のチップの最大定格を超えてしまいます．このため，電流制限抵抗が必要となります．電流制限抵抗をコモン側に挿入すると，発光するセグメントの数により輝度が変化し，見苦しくなります．

カソード

Cathode．陰極．電子デバイスにおいてマイナスの電圧を印加する電極のことを指す．

電流制限用抵抗

負荷に対して必要以上の電圧を印加しようとする場合に，直列に抵抗を挿入して不必要な電圧を吸収し，負荷に定格以上の電流が流れないように処置をする．この場合に回路に挿入する抵抗のことを電流制限用抵抗と呼ぶ．

〈リスト4-11〉7セグメント・デコーダ(カソード・コモンLED用)

```
----------------------------------------------------------------
-- 7-segment decoder
----------------------------------------------------------------
--
--   segment layout (alphabet appear segment's name.)
--
--              A
--          F       B
--              G
--          E       C
--              D
--
library ieee;
use ieee.std_logic_1164.all;

entity dec7Seg is
             port(d     : in std_logic_vector(3 downto 0);
                  segA : out std_logic;
                  segB : out std_logic;
                  segC : out std_logic;
                  segD : out std_logic;
                  segE : out std_logic;
                  segF : out std_logic;
                  segG : out std_logic);
end dec7Seg;

architecture rtl of dec7Seg is

          signal segBuf : std_logic_vector(6 downto 0);

begin
          with d select
                segBuf <=     "0111111" when "0000",     -- '0'
                              "0000110" when "0001",     -- '1'
                              "1011011" when "0010",     -- '2'
                              "1001111" when "0011",     -- '3'
                              "1100110" when "0100",     -- '4'
                              "1101101" when "0101",     -- '5'
                              "1111101" when "0110",     -- '6'
                              "0000111" when "0111",     -- '7'
                              "1111111" when "1000",     -- '8'
                              "1101111" when "1001",     -- '9'
                              "0000000" when others;

          segG <= segBuf(6);
          segF <= segBuf(5);
          segE <= segBuf(4);
          segD <= segBuf(3);
          segC <= segBuf(2);
          segB <= segBuf(1);
          segA <= segBuf(0);

end rtl;
```

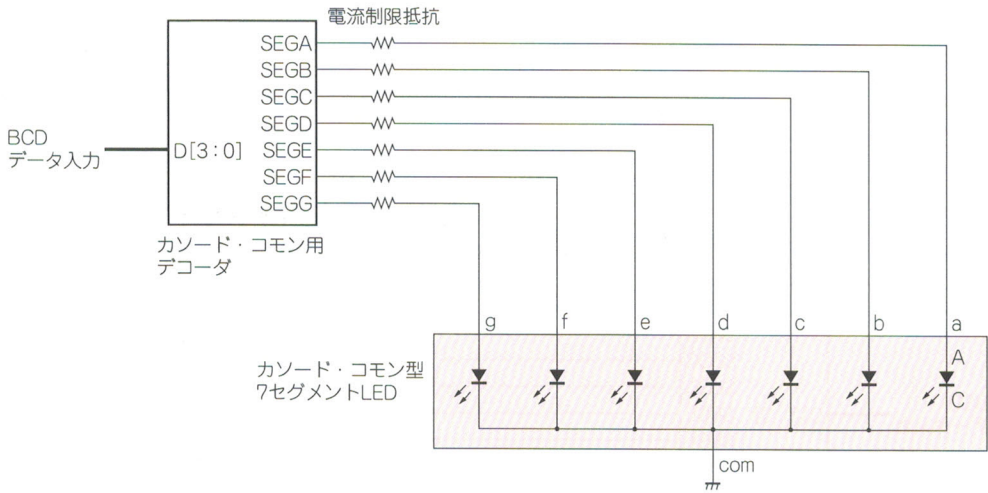
駆動するチップ側の電源電流の最大値にも気を付けましょう(LEDへの駆動電流は電源端子より供給される).

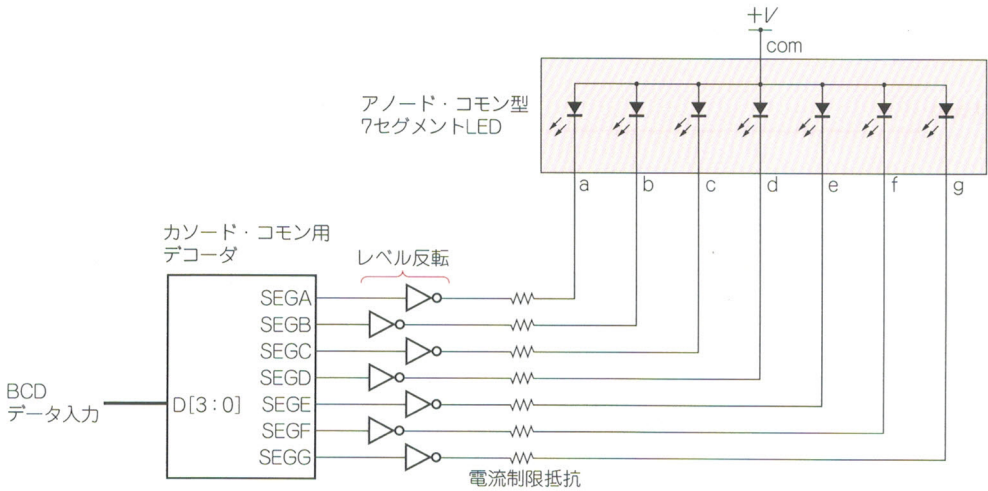
アノード・コモンLEDの駆動レベルは，カソード・コモンLEDをドライブする場合と正反対の値となります．このため，カソード・コモン用のデコーダの出力を反転するか，データ・テーブルを反転したアノード・コモン用のデコーダを作るなどして対応します(**図4-32**).

LCD(液晶ディスプレイ)は，内面側に透明電極を形成した2枚のガラス板の間に液晶物質を封入した構造をもちます．片方のガラス内面に施された特殊な処理により，電界が加わっていないときには，液晶の結晶は一定方向に整列しています．しかし，液晶に電界を加えると液晶の並び方にねじれが生じ，電界が加わっ

偏光性

自然界に存在する光の波は，いろいろな方向に振動をしているが，偏光板を通すと特定の方向に振動している光のみを選択的に通過させることができる．液晶物質内の結晶も偏光板のような偏光作用をもち，かつ印加される電界の強さによりその偏光角度が変化する．

〈図4-31〉カソード・コモンLEDの駆動

備考）カソード・コモン型LEDにおいては，セグメント出力が1のときにセグメントが点灯する

〈図4-32〉アノード・コモンLEDの駆動

備考）アノード・コモン型LEDにおいては，セグメント出力が0のときにセグメントが点灯する

ていない場合とくらべ**偏光性**が変わります．液晶のこのような性質を利用して，偏光板と組み合わせることにより，電界の印加の有無により光の透過率が変化する光学デバイスができるわけです．これを数字などの表示に利用したものがLCDです．

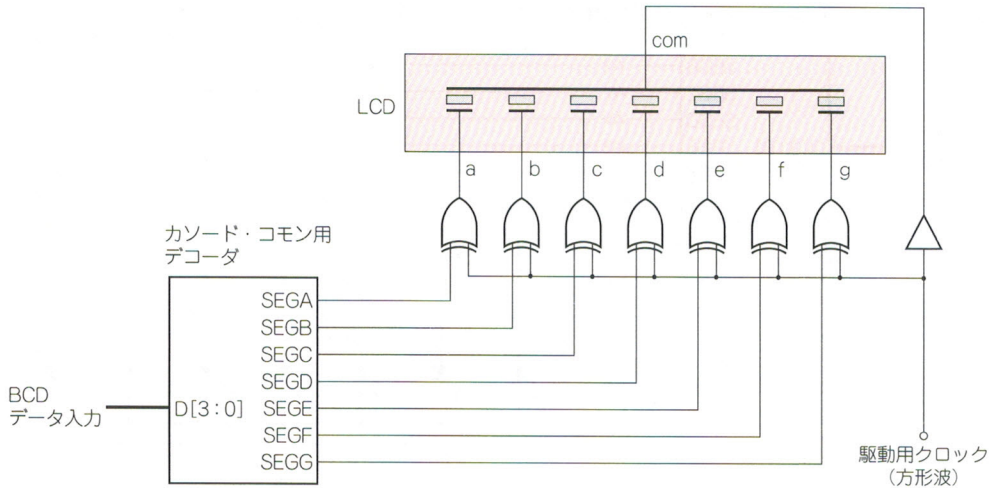
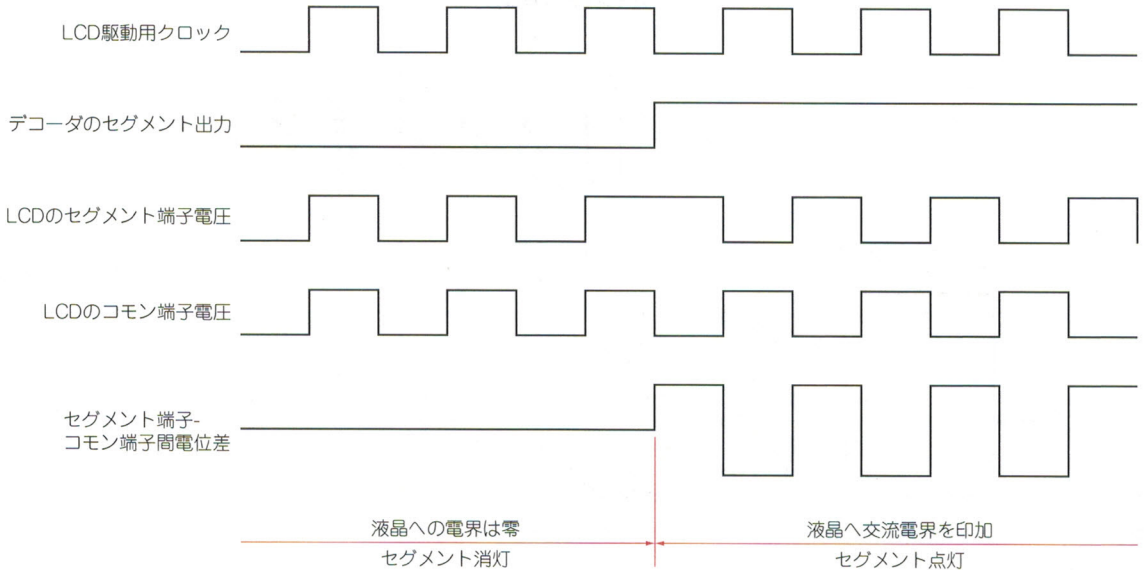
電界を加えればよいのならば，7セグメント・デコーダで直接ドライブできるのかというと，話はそう単純ではありません．液晶も一応液体ですから，長時間直流電界を印加し続けると電気分解が起こり，**液晶物質が劣化**してしまうのです．メーカの保証寿命をまっとうするためには，液晶を交流駆動してやらなくてはなりません．

そこで登場するのが**Ex-ORゲート**です．Ex-ORは，片方の入力が0の場合にはもう片方の入力の値をそのまま出力に伝え，また片方の入力が1の場合にはもう片方の入力の値を反転して出力に伝達するという性質をもっています．

これを利用し，液晶駆動用のクロックをドライバを介してLCDのコモン端子

液晶物質が劣化
筆者も劣化したLCDの現物を拝んだことがないので明確に言いきることはできないが，コントラストが低下したりするのではないかと思われる．気泡が生ずることもあるのだろうか？

Ex-ORゲート
排他的論理和演算を行う論理ゲートのこと．2本の入力が一致しない場合に出力が‘1’となる．

〈図4-33〉 単相LCDの駆動

〈図4-34〉 LCD駆動電圧

に印加し，LCDのセグメント端子には，駆動用クロックとデコーダのセグメント出力とのEx-OR結果を加えるようにします(**図4-33**)．

このようにすると，デコーダのセグメント出力が0の場合には，LCDのコモン端子とセグメント端子の波形が同相となり，端子間の電位差はゼロになります．デコーダのセグメント出力が1の場合には，LCDのコモン端子とセグメント端子の波形が逆相となり，端子間にドライバの出力電圧の2倍の電圧の方形波が印加されます(**図4-34**)．

LCDのコントラストは液晶に印加される電界の強さと相関関係にあるため，コントラストを調整するためには，駆動用ドライバの出力電圧が変化できなければなりません．

なお，**単相LCDの駆動用クロック**には，数百Hz程度の周波数が用いられるようです．

単相LCDの駆動用クロック

マルチプレクス（多重化）が行われていない単純なLCD表示器の場合には，単相のクロックで駆動することが可能．多重化が行われているLCDの駆動には，多相の多値レベルをもつ複雑な波形の駆動信号が必要になる．このため，多重化が行われているLCDの駆動には専用のドライバICが使われる．

プライオリティ・エンコーダの記述法

デコーダとは逆に，それぞれに**重み**の異なる複数の入力信号があり，そのうちの1本がアクティブになった場合に，その入力のもつ重みをコード(たとえば2進数)で返すのがエンコーダ回路です．

しかし，同時に2本以上の入力がアクティブとなってはならない，などという条件が付いては回路として使いづらいため，入力信号に優先順位を付け，複数の入力がアクティブになった場合にはそれにしたがって，ただ1本だけの入力のみを有効とすることにしたもの．それがプライオリティ・エンコーダです．

4入力のプライオリティ・エンコーダの記号と真理値表を**図4-35**に示します．

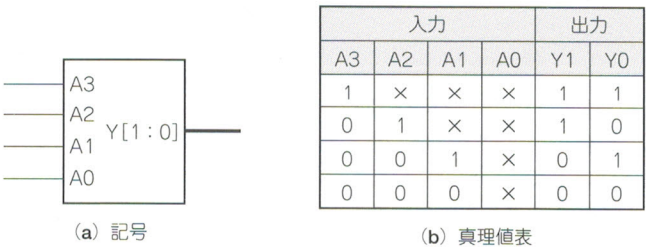
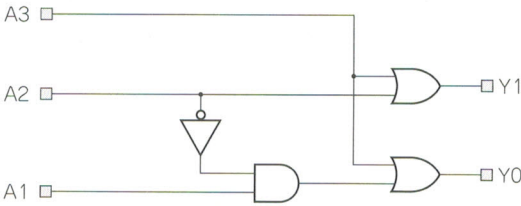
重み

プライオリティ・エンコーダにおいては，それぞれの入力に数値的な重みが割り当てられている．入力信号がアクティブとなった場合には，その入力がもつ重みに対応するコード(2進数)を出力する．

〈図4-35〉4入力のプライオリティ・エンコーダ

(a) 記号

入力				出力	
A3	A2	A1	A0	Y1	Y0
1	×	×	×	1	1
0	1	×	×	1	0
0	0	1	×	0	1
0	0	0	×	0	0

(b) 真理値表

〈図4-36〉4入力のプライオリティ・エンコーダ回路

〈リスト4-12〉4入力のプライオリティ・エンコーダ

```
--
-- 4-to-2 priority encoder
--
library ieee;
use ieee.std_logic_1164.all;

entity pEnc2 is
            port(   a3  : in std_logic;
                    a2  : in std_logic;
                    a1  : in std_logic;
                    a0  : in std_logic;
                    y   : out std_logic_vector(1 downto 0));
end pEnc2;

architecture rtl of pEnc2 is
begin

            y <=    "11" when (a3 = '1') else
                    "10" when (a2 = '1') else
                    "01" when (a1 = '1') else
                    "00";

end rtl;
```

この場合には，A3入力のプライオリティをもっとも高いものとしています．このような機能を実現する回路は**図4-36**のようなものとなります．

プライオリティ・エンコーダを作ろうとする場合，ゲートによる回路構成を考えて，**論理演算子で記述する**こともできますが，VHDLにおいては多条件のwhen ～ else文を使って機能記述をしてしまったほうが設計が容易です．多条件のwhen ～ else文においては，先に書かれた条件より順に判断がなされていくため，プライオリティの高い入力の条件判断より順に記述していけば，簡単に優先順位を決定することができます．

既製(標準ロジック)のプライオリティ・エンコーダでは，データの重みが大きい入力の優先順位が高くなっていたように記憶していますが，プログラマブルなデバイスを用いる場合などには，かならずしもこれに習う必要はありません．優先順位は自由に決めることが可能なので，必要に応じて変更してもよいわけです(**リスト4-12**)．

論理演算子で記述する
ディジタル回路の記述をする場合，たいていの回路は論理演算子で記述することが可能．また，論理演算子で記述されたコードは確実に回路合成することが可能．

ROMの記述法

小さなROMであれば，通常のCPLDにも**インプリメント**可能ですし，**FLEXシリーズ**の搭載RAM(EAB)はROMとして運用することも可能です．

ここでは例として，**2乗テーブル**を作ってみましょう．4ビット・データの2乗テーブルは**表4-6**のようになります．

テーブルROMの場合には，このテーブル自体が真理値表と一致します．

ここでは，各アドレスに対応する信号にいったん定数を代入し，with ～ select文により与えられたアドレスにしたがって信号を選択するセレクタを記述することによりROMを実現しています(**図4-37**，**リスト4-13**)．

インプリメント
Implement．履行するとか，(条件，要件)を満足するなどの意味．ここでは，VHDLで書きあらわした構造/機能記述をチップ上で実現する(チップ上に展開する)という意味をあらわしている．

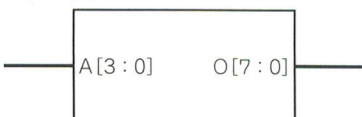
FLEXシリーズ
アルテラ社のCPLDシリーズの一つ．SRAMベースで通常のCPLDより単位回路(マクロセル)の粒度が小さく，規模の大きいデバイスが揃っている．チップ上にSRAMを搭載しており，これはROMとしても使える．

2乗テーブル
数値をあたえると，それを2乗した値を返す数表のこと．ここではROMを使って実現している．

〈表4-6〉2乗テーブル(入力：4ビット，出力：8ビット)

入力（アドレス）		出力（データ）	
10進	2進	2進	10進
15	1 1 1 1	1 1 1 0 0 0 0 1	225
14	1 1 1 0	1 1 0 0 0 1 0 0	196
13	1 1 0 1	1 0 1 0 1 0 0 1	169
12	1 1 0 0	1 0 0 1 0 0 0 0	144
11	1 0 1 1	0 1 1 1 1 0 0 1	121
10	1 0 1 0	0 1 1 0 0 1 0 0	100
9	1 0 0 1	0 1 0 1 0 0 0 1	81
8	1 0 0 0	0 1 0 0 0 0 0 0	64
7	0 1 1 1	0 0 1 1 0 0 0 1	49
6	0 1 1 0	0 0 1 0 0 1 0 0	36
5	0 1 0 1	0 0 0 1 1 0 0 1	25
4	0 1 0 0	0 0 0 1 0 0 0 0	16
3	0 0 1 1	0 0 0 0 1 0 0 1	9
2	0 0 1 0	0 0 0 0 0 1 0 0	4
1	0 0 0 1	0 0 0 0 0 0 0 1	1
0	0 0 0 0	0 0 0 0 0 0 0 0	0

〈図4-37〉2乗テーブルROMのシンボル

A[3：0]　O[7：0]

〈リスト4-13〉2乗テーブルROM

```
----------------------------------------------------------------
-- square table rom
----------------------------------------------------------------
library ieee;
use ieee.std_logic_1164.all;

entity romSq is
        port    (a  : in std_logic_vector(3 downto 0);
                 o  : out std_logic_vector(7 downto 0));
end romSq;

architecture rtl of romSq is

        signal data15 : std_logic_vector(7 downto 0);
        signal data14 : std_logic_vector(7 downto 0);
        signal data13 : std_logic_vector(7 downto 0);
        signal data12 : std_logic_vector(7 downto 0);
        signal data11 : std_logic_vector(7 downto 0);
        signal data10 : std_logic_vector(7 downto 0);
        signal data9  : std_logic_vector(7 downto 0);
        signal data8  : std_logic_vector(7 downto 0);
        signal data7  : std_logic_vector(7 downto 0);
        signal data6  : std_logic_vector(7 downto 0);
        signal data5  : std_logic_vector(7 downto 0);
        signal data4  : std_logic_vector(7 downto 0);
        signal data3  : std_logic_vector(7 downto 0);
        signal data2  : std_logic_vector(7 downto 0);
        signal data1  : std_logic_vector(7 downto 0);
        signal data0  : std_logic_vector(7 downto 0);

begin

--                                        address / data
        data15 <= "11100001";           -- 0Fh     E1h...225d
        data14 <= "11000100";           -- 0Eh     C4h...196d
        data13 <= "10101001";           -- 0Dh     A9h...169d
        data12 <= "10010000";           -- 0Ch     90h...144d
        data11 <= "01111001";           -- 0Bh     79h...121d
        data10 <= "01100100";           -- 0Ah     64h...100d
        data9  <= "01010001";           -- 09h     51h... 81d
        data8  <= "01000000";           -- 08h     40h... 64d
        data7  <= "00110001";           -- 07h     31h... 49d
        data6  <= "00100100";           -- 06h     24h... 36d
        data5  <= "00011001";           -- 05h     19h... 25d
        data4  <= "00010000";           -- 04h     10h... 16d
        data3  <= "00001001";           -- 03h     09h...  9d
        data2  <= "00000100";           -- 02h     04h...  4d
        data1  <= "00000001";           -- 01h     01h...  1d
        data0  <= "00000000";           -- 00h     00h...  0d

        with a select
                o <= data15 when "1111",
                     data14 when "1110",
                     data13 when "1101",
                     data12 when "1100",
                     data11 when "1011",
                     data10 when "1010",
                     data9  when "1001",
                     data8  when "1000",
                     data7  when "0111",
                     data6  when "0110",
                     data5  when "0101",
                     data4  when "0100",
                     data3  when "0011",
                     data2  when "0010",
                     data1  when "0001",
                     data0  when "0000",
                     "00000000" when others;

end rtl;
```

第5章 データを記憶する機能をもつ フリップフロップ(レジスタ)の記述

吉澤　清

電子回路を設計する場合に必要となるもう一つの要素は，データを記憶する機能をもつ**フリップフロップ**(レジスタ)です．本章では，このフリップフロップの記述に関して解説します．

フリップフロップ
1ビットのデータの記憶が可能で，複数の制御入力によりそのデータを制御することが可能なディジタル回路の基本回路のこと．

フリップフロップの記述には**プロセス文**を用います．しかし，プロセス文による記述はVHDLの中でもいちばん多様性に富んでいて，憶えなければならないことも多く，マスタするのがたいへんなところです．

プロセス文
process文．VHDLの基本構文の一つ．フリップフロップの記述には不可欠．

そこで本章では，プロセス文に関する内容のうち，フリップフロップの記述に必要なことがらだけをダイジェスト的にまとめることにしました．本章で新しく登場するのは，

▶プロセス文
▶if ～ then ～ else文
▶'eventアトリビュート

の3点だけです．

プロセス文を用いた記述に関してさらに詳しく知りたいという方は，次号も参照してください．

プロセス文の働きと記述のしかた

プロセス文は，機能モジュール内のアーキテクチャ部に記述します．構文は**図5-1**に示すようなものです．

まず，プロセス文には，各プロセスの区別をつけるためにラベルを付けることができます(オプション)．processに続くカッコ内には信号名(複数可)を記入します．ここに書かれた信号名は**センシティビティ・リスト**と呼ばれ，process文はここに書かれた信号の変化を検出し，いづれかの信号に変化が生じた瞬間にprocess以降のbegin ～ end process; 間に記述された内容を**ワンショットで実行**します．まあ，回路にたとえるならば，プロセス文は微分回路のようなものと言えます(**図5-2**)．

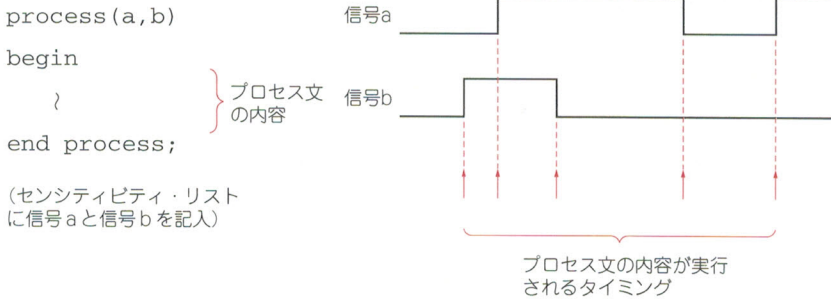
センシティビティ・リスト
プロセス文起動のトリガとなる，変化検出用の信号リスト．

ワンショットで実行
プロセス文内の処理はシーケンシャルに記述されることもあるが，すべての処理はセンシティビティ・リストの信号の変化の検出にともない,瞬間的に実行される．ただし，実回路化した場合にはそれなりの遅延を伴う．

また，プロセス文の中ではwhen ～ else文とwith ～ select文を使うことはできません(エラーとなります)．もし，when ～ else文やwith ～ select文が必要な場合には，プロセス文の外に書くようにします．

〈図5-1〉プロセス文の構文

```
ラベル：process (センシティビティ・リスト)
        begin
          ～
        end process;
```

〈図5-2〉プロセス文の働き

if 〜 then 〜 else文の働きと記述のしかた

if 〜 then 〜 else文は，条件式の判定結果により，特定の処理を実行したり，実行させなかったりする機能をもった構文です．if 〜 then 〜 else文はファンクションまたはプロセス文の中でのみ使うことができます．プロセス文の外で記述すると**エラー**になります．

if 〜 then 〜 else文の構成要素は，つぎの四つの文節です(**表5-1**)．

▶ if (条件式) then
▶ end if;
▶ else
▶ elsif (条件式) then

if 〜 then 〜 else文の記述はif (条件式) thenで始まり，end if; で終わります．elseおよび elsif (条件式) thenは，if (条件式) thenとend if; の間に書かなければなりません．

文章で説明してもわかりづらいので，if 〜 then 〜 else文のいくつかのパターンについて見ていきましょう．

始めは，if (条件式) thenと end if; だけの場合です(**図5-3**)．条件式が成立したときには処理(then 〜 end if間に書かれた記述)が実行されますが，不成立の場合は何も実行されません．**条件により処理のオン/オフが可能**なわけです．

つぎはelse文の登場です(**図5-4**)．条件式が成立したときには処理Aが実行さ

エラー

if〜then〜else文やcase文はプロセス文やファンクションの中でしか使えない．プロセス文の外で使うと，VHDLシミュレータでも回路合成ツールでもエラーとなる．

条件により処理のオン/オフが可能

この機能を使うことにより，入力のレベルや信号の変化などの条件が成立したときにだけ，特定の処理を行うような回路を実現することができる．

〈表5-1〉四つの文節の意味

if (条件式) then	もし，条件式が成立したら次の文節までの処理を実行する．
end if;	if〜then〜else文の末尾を表す．VHDLにおいてはif〜then〜else文でスイッチする処理が複数行に及ぶことが許されているため，構文の末尾を明示することが必要．
else	if (条件式) then節およびelsif (条件式) then節と対となるelse節は，else節に先行するif (条件式) then節またはelsif (条件式) then節の条件式が成立しなかった場合に，else以降end if;までの処理を実行する．
elsif (条件式) then	先行するif (条件式) then節またはelsif (条件式) then節の条件式が成立しなかった場合に，次の新たな条件判断を行う．新たな条件式が成立したら次の文節までの処理を行う．elseifではなくelsifであるので，タイプ時に綴りに注意．

〈図5-3〉if（条件式）then と end if; だけの場合

〈図5-4〉else が入った記述

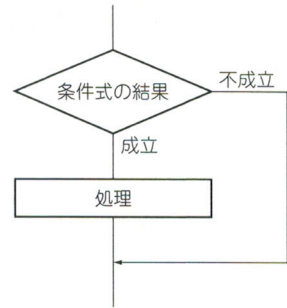
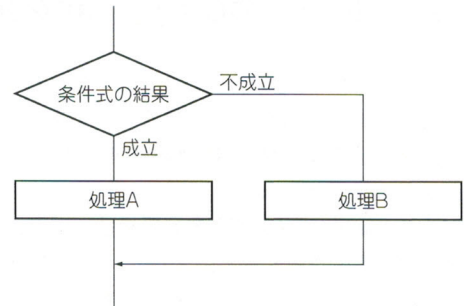
〈図5-5〉elsif を使った記述

れ，条件式が不成立の場合には処理Bが実行されます．条件により処理のスイッチ（切りかえ）が可能なわけです．

elsifを使った**図5-5**のような記述の場合においては，条件式1が成立したときには処理Aが実行されます．条件式1が不成立の場合には条件式2の判断へと進みます．そして，条件式1が不成立，条件式2が成立の場合は処理Bが，条件式1が不成立，条件式2も不成立の場合は処理Cが実行されます．

この記述をelsifを使わないで書くと**図5-6**のようになります．

elsifというのは，プログラム言語にもない語彙のようですが，elsifがあることにより，VHDLにおいてはフリップフロップの記述が簡潔にまとめられます．

〈図5-6〉elsifを使わない書き方

```
if (条件式1) then
        [処理A]
else
    if (条件式2) then
            [処理B]
    else
            [処理C]
    end if;
end if;
```

'eventアトリビュートの働き

アトリビュート(属性)とは，信号や変数がどんな状態であるかとか，どんなフォーマットをもつかを表す**データの付随情報**のことです．

'eventアトリビュートはプロセス文の中で信号の変化が検出された場合にtrueとなります． 'eventアトリビュートは，フリップフロップの記述においてc'eventなどの形でクロックの変化の検出のために用いられます．

フリップフロップの記述に必要な要素が揃ったので，実際のフリップフロップの記述について紹介します．ここで例として取り上げるのは，フリップフロップの中でももっともポピュラな，非同期リセット付きDフリップフロップです．

データの付随情報
フリップフロップの動作を記述する場合には，クロックの変化を検出する必要がある．'eventアトリビュートによればデータが変化したかどうかの情報を抽出することが可能．

非同期リセット付きDフリップフロップの記述

図5-7は**非同期リセット付きDフリップフロップ**の記号と真理値表です．

図5-7のフリップフロップの記述は，**図5-8**のようになります．

それではコードについて解説します．まず，お決まりの**スタンダード・ロジック型のライブラリ**を使用する旨の宣言があります．

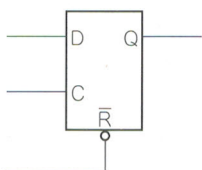
エンティティ部の記述は，この回路がd，c，nr(負論理のrの意)の3本のstd_logic型の入力信号と，qというstd_logic型の出力信号をもつことを示しています．

アーキテクチャ部に移って，今回はとくに内部信号は必要としていないので，

非同期リセット付きDフリップフロップ
Dフリップフロップは，クロックの立ち上がりでデータを取り込む動作を行うフリップフロップ．非同期リセットは，クロックの変化に関係なくリセット信号のレベルによりフリップフロップのデータをリセットできる機能のこと．

スタンダード・ロジック型のライブラリ
VHDLで回路を設計する場合に標準的に使用するデータ型である，std_logic型とstd_logic_vector型を定義しているライブラリ．VHDLの処理系ではこれら二つの型について標準ではサポートしておらず，機能モジュールを記述するたびに，このライブラリの使用を宣言しなければならない．

〈図5-7〉非同期リセット付きDフリップフロップ

(a) 記号

入力			出力	動作
R̄	C	D	Q	
0	×	×	0	リセット
1	↑	1	1	データの取り込み
1	↑	0	0	
1	↓	×	Qn	保持

(b) 真理値表

〈図5-8〉非同期リセット付きDフリップフロップのVHDL記述

```
library ieee;
use ieee.std_logic_1164.all;

entity df is
   port( d  : in std_logic;
         c  : in std_logic;
         nr : in std_logic;
         q  : out std_logic);
end df;

architecture rtl of df is
begin

   process(nr,c)
   begin

      if (nr = '0') then
         q <= '0';
      elsif (c'event and c = '1') then
         q <= d;
      end if;

   end process;

end rtl;
```

エンティティ部
アーキテクチャ部
プロセス文
if～then ～else文
'event アトリビュート

signal文による内部信号の宣言はありません．アーキテクチャ部にはプロセス文があります．真理値表を見てわかるように，非同期リセット付きDフリップフロップはnrとc入力の変化点においてアクションを行います．このため，プロセス文のセンシティビティ・リストには，nrとcを並記しています．結果として，プロセス文内の処理(この場合は一つのif ～ then ～else文)は，nrまたはcの信号の変化点において**ワンショット的に実行**されます．

フリップフロップの機能記述の核心部は，プロセス文内の if ～ then ～ else文です．if ～ then ～ else文における第一の条件判断は，非同期リセット付きDフリップフロップでいちばん優先度の高い入力であるnrに関するものです．真理値表を見ると，$\overline{\mathrm{R}}$(nr)入力が‘0’の場合にはc入力やd入力の値にかかわらずq出力が‘0’となっています．このため，if ～ then ～ else文の1，2行目では，nrが‘0’である場合に信号qに‘0’を代入するように記述しています．

もし，nr入力が‘0’以外の場合(つまり‘1’の場合)には，第二の条件判断へと進みます．真理値表によれば，nr入力が‘1’の状態(**リセットが外れている**)において，クロックの立ち上がりのタイミングでデータが取り込まれています．elsif文を使用することにより，nr入力が‘0’でない場合という条件付けはすでになされているため，elsifにより判断する条件は，クロックの立ち上がりということになります．elsif に続くカッコ内の

```
c'event and c = '1'
```

が，クロックの立ち上がりを示しています．c'eventがc(クロック)の変化を，c ＝‘1’がクロックが‘1’であることを示しています．条件は両者のANDとなっているので，クロックが変化し，その結果として‘1’となった場合，つまりクロックの立ち上がりを意味します．この条件が成立した場合にのみd入力がq出力へと代入されます．さらにそれ以外の場合に関しては，続くelse文の記述がないため，何も行われません．何も行わないということは**状態を保持**するとい

ワンショット的に実行
プロセス文内の処理はシーケンシャルに記述されることもあるが，すべての処理はセンシティビティ・リストの信号の変化の検出にともない，瞬間的に実行される．ただし，実回路化した場合にはそれなりの遅延を伴う．

リセットが外れている
nr入力は負論理なので，‘1’レベルのときに非アクティブとなる．

状態を保持
記憶を含んだ回路において，記憶している情報が変化せずに，そのままの状態を保つこと．

うことを示しています．

以上のように，VHDLにおけるフリップフロップの記述は，回路の構造を書き表すのではなく，回路がどう働くかを表現する形になります．

'event アトリビュートの必要性

どうせ，プロセス文自体が信号の変化の検出を行ってくれるのに，なぜわざわざ 'eventアトリビュートなどというものが必要なんだ，と疑問に思う方はいないでしょうか．

プロセス文のセンシティビティ・リストの中身が，一つの信号だけの場合は，'eventアトリビュートがなくても何ら問題はありません．たとえば，**非同期入力**を何ももたないDフリップフロップの場合は，**図5-9**のような記述でも実現することができます．この場合には，プロセス文でクロックの変化を検出し，if ～ then～else文でクロックのレベルが '1' であることを検出するため，トータルでクロックの立ち上がりを判定しているわけです．

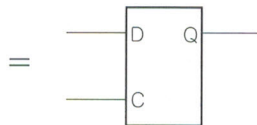
[注意]**図5-9**の書き方でもDフリップフロップの回路合成は可能です．ただし，このような場合だけを特別扱いにするのも煩雑なのでフリップフロップを記述する場合はやはり 'event を用いた書き方に，{if (c'event and c = '1') then というように} 統一することをおすすめします．

'eventアトリビュートがないと問題になるのは，プロセス文のセンシティビティ・リストの中に，複数の信号が書かれている場合です．たとえば，非同期リセット付きDフリップフロップを記述をしようとして，**図5-10**のような記述を行うと，プロセス文はクロックの変化点以外にリセット(nr)の変化点をも検出してしまうため，

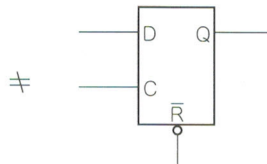
非同期入力
クロックとは無関係に機能する入力信号のこと．たとえば，非同期リセット入力や非同期セット入力など．

〈図5-9〉非同期入力をもたないDフリップフロップ

```
process(c)
begin
  if (c = '1') then
    q <= d;
  end if;
end process;
```

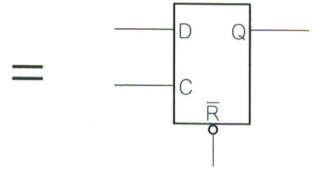
＝ (D, Q, C)

〈図5-10〉非同期リセット付きDフリップフロップを記述しようとして…

```
process(nr,c)
begin
  if (nr = '0') then
    q <= '0';
  elsif (c = '1') then
    q <= d;
  end if;
end process;
```

≠ (D, Q, C, $\overline{R}$)

〈図5-11〉非同期入力をもつDフリップフロップ

```
process(nr,c)
begin
  if(nr = '0') then
    q <= '0';
  elsif(c'event and c = '1') then
    q <= d;
  end if;
end process;
```

=

D Q
C
R̄

```
elsif (c = '1') then
```

の文節では，クロックの立ち上がりだけでなく，クロックが'1'でありかつリセット信号が変化したときにもデータを取り込むことになってしまいます．一般的に，Dフリップフロップは，クロックが'1'レベルのときにリセットの立ち上がりエッジや立ち下がりエッジでデータを取り込んだりはしないので，これでは**適切なモデリング**とは言えません．

このため，非同期入力をもつDフリップフロップを記述する場合には，**図5-11**に示すように，'eventアトリビュートを用いてクロックの立ち上がりを明確に示す必要があります．

適切なモデリング
VHDLにおいては，フリップフロップは機能記述により書かれる．書かれた記述がフリップフロップの動作と一致していない場合，その記述は適切なモデルとは言えない．

elseの記述ができない場合

if（条件式）thenまたは，elsif（条件式）then文節において，条件式の内容が信号の変化の検出であった場合，このif（条件式）thenまたはelsif（条件式）thenに対応するelse文節を記述すると，シミュレーションは可能であったとしても，回路合成時に**合成不能**となります．

信号の変化を条件とする条件判断が使われる場合としては，フリップフロップの記述におけるクロックのエッジ検出があります．たとえばDフリップフロップの記述は，**図5-12**のようになります．この場合のif（条件式）then文節の条件は(c'event and c = '1')，つまりクロックの立ち上がりです．この記述では，if （条件式）thenに対応するelseの記述がないため，何ら問題はありませんが，これにelse文節が加わると回路合成ツールが受けつけてくれなくなります．

これは実回路において，クロックの立ち上がりおよび立ち下がりの両エッジで動作するフリップフロップが実現できないためであると思われます．大昔の設計

合成不能
現状では，クロック入力の両エッジで動作を行う実用的なフリップフロップは存在しない．回路で実現できない機能をVHDLで記述した場合，単体VHDLシミュレータ上でシミュレーションを行うことは可能だが，回路合成ツールが実際の回路に落とすことはかなわない．

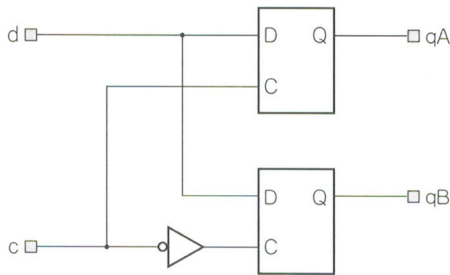
〈図5-12〉信号の変化を条件とするDフリップフロップ

```
process(C)
begin
 if (c'event and c = '1') then
  q <= d;
 end if;
end process;
```

〈図5-13〉合成できない記述の例

```
process(C)
begin
 if (c'event and c = '1') then
  qA <= d;
 else
  qB <= d;
 end if;
end process
```

if（信号の変化の条件）thenに対応するelse文節

〈図5-14〉図5-13の記述の回路(クロックの立ち上がりで動くフリップフロップと立ち下がりで動くフリップフロップ)

の仕方としては，クロックを両側微分してフリップフロップのクロックに与えることにより，クロックの両エッジで作動させるなどという芸当もあったようです．しかし，これはディスクリートで低速で働く回路を組んでいた時代の話であり，今日では通用しません．ICの世界にこのような技術を持ち込もうとした場合，何回も繰り返し試作を行わねばならず開発費が膨大なものとなり，およそ**商売としては成立しません**．

合成できない記述の例としては，たとえば**図5-13**のようなものもあります．この記述は，**図5-14**のような回路をVHDLで表現したものです．

この回路は，両側エッジで働くフリップフロップを記述したものではなく，クロックの立ち上がりで動くフリップフロップとクロックの立ち下がりで動くフリップフロップを一つのプロセス文にまとめたものです．この記述が合成不能であるということは，おそらく回路合成ツールは回路合成の結果両側エッジ駆動となった場合にエラーとしているのではなく，エッジ検出を行っている if　(条件式) thenに対応するelse文節の存在をもって合成不能の判定をしているのではないかと思われます．

したがって，**逆相クロックで動くフリップフロップ**を記述する場合には，プロセス文を分けて書く必要がありそうです．「エッ？　逆相クロックは使ってはいけないのでは」と疑問を感じる方もいることでしょう．当然，初心者の方には危険なのでおすすめはしません．しかし，百戦錬磨の回路設計者にそんな野暮なことは言えないでしょう．そういった設計をしないと実現できない回路も存在するのですから．回路の裏の裏まで熟知した人が，**オウンリスク**できちんとタイミング設計を行って実現するのであれば，何人もそれを止めることはできません．アルテラ社製のCPLDは，非同期回路にも対応しているようですし….

ただし，この辺りになると誰もができるというレベルの話ではありません．下手に手を出すと火傷をします(筆者などは怖くて手が出せません)．

商売としては成立しません

特性的にクリティカルな回路をICチップ上で実現しようとする場合，所望の特性が得られるまで，パターンを修正しながら試作を繰り返す必要がある．今日では，プロセスの複雑化にともない，チップの試作にかかる費用も高額になっており，よほど特別な(システムの根幹を支えるような)機能でなければ，このような時間(＝人件費)と費用のかかる作業はペイしない．

逆相クロックで動くフリップフロップ

基本的にクロックの立ち上がりでフリップフロップが動作しているシステムに，逆相クロック(クロックの立ち下がり)で動作するフリップフロップを混ぜて回路を構成すると，部分的にフリップフロップの間に入る回路の伝搬遅延の制約が厳しく(許容される伝搬遅延が半分に)なり，またクロックのデューティ比の変動を考慮すると，その度合いはさらに高まる．このような制約を考慮せずに逆相クロックを使うと，期待した速度が得られないなどのトラブルの原因となる．

オウンリスク

own risk(自己責任)．設計に関する自分の判断が回路のできを左右する回路設計の世界は，完全に自己責任の世界．

出力バッファの必要性

機能モジュールの**入出力信号の方向性**を出力(out)に指定すると，機能モジュール内の信号代入文でその出力信号(たとえばq)に値を与えることはできますが，機能モジュールの中から出力信号(q)の値を参照することはできなくなります．

このため，出力信号のフィードバックを伴うフリップフロップ，レジスタ，カウンタなどの記述を行おうとする場合には，バッファ用の信号(たとえばbufQ)を使って回路の記述を行い，最後にバッファ用の信号を出力信号に代入するようにします．

それでは，いろいろなフリップフロップのVHDL記述について紹介します．

入出力信号の方向性

機能モジュールの入出力信号に関しては，入力，出力，入出力のいづれかの方向性を指定する．

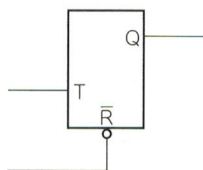
非同期リセット付きTフリップフロップの働き

リプル・カウンタ
T型フリップフロップをシリーズに接続したカウンタ．同期カウンタとくらべると回路が簡単で，トグル周波数も上げられ，消費電力も少なくなる傾向をもつ．しかし，クロック入力から出力までの伝搬遅延は後段になるほど大きくなり，出力をデコードした場合にグリッチが出たりする．

クロックが立ち上がるたびに出力が反転するTフリップフロップは，**リプルカウンタ**(非同期カウンタ)の構成要素として広く用いられていました．

非同期リセット付きTフリップフロップの記号と真理値表を**図5-15**に示します(**リスト5-1**)．

〈図5-15〉非同期リセット付きTフリップフロップ

(a) 記号

入力		出力	動作
R̄	T	Q	
0	×	0	リセット
1	↑	Q̄n	反転
1	↓	Qn	保持

(b) 真理値表

〈リスト5-1〉非同期リセット付きTフリップフロップ

```
--
-- T-F.F. with asynchronous reset
--
library ieee;
use ieee.std_logic_1164.all;

entity tf is
          port(   t   : in std_logic;
                  nr  : in std_logic;
                  q   : out std_logic);
end tf;

architecture rtl of tf is

          signal bufQ  : std_logic;

begin

          process(nr,t)
          begin

                  if (nr = '0') then
                         bufQ <= '0';
                  elsif (t'event and t = '1') then
                         bufQ <= not bufQ;
                  end if;

          end process;

          q <= bufQ;

end rtl;
```

J-Kフリップフロップの働き

J-Kフリップフロップの記号と真理値表を**図5-16**に示します．

シーケンス制御回路
シーケンサ回路のこと(第8章に記述例あり．ただし，本書で取り上げているのは，Dフリップフロップを用いたワンホット・シーケンサ)．

データ入力をもたないJ-Kフリップフロップは，標準ロジックICで回路を組んでいた時代の同期設計用フリップフロップです．カウンタや**シーケンス制御回路**などに，幅広く使われていました．

CPLDやFPGAの基本回路はDフリップフロップですが，**図5-17**のようにゲート回路でフィードバックをかけることにより，J-Kフリップフロップを実現することが可能です(**リスト5-2**，**リスト5-3**)．

同期イネーブル付きDフリップフロップ(同期設計用Dフリップフロップ)の働き

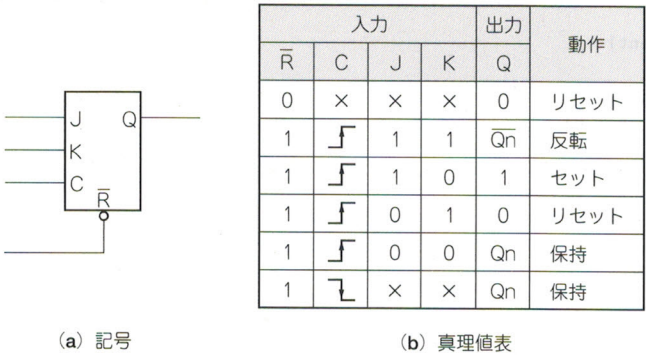
システム・クロック
ICチップ全体を駆動するクロック信号のこと．

同期設計においては，すべてのフリップフロップ，レジスタ，カウンタのクロック入力が**システム・クロック**に接続されますが，これにより困ったことが起こります．Dフリップフロップが単なる**ディレイ回路**になってしまうのです．Dフ

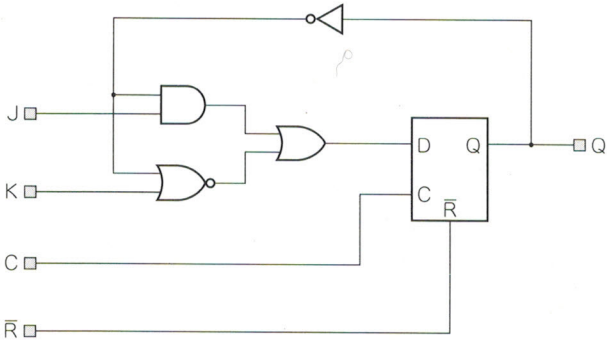
〈図5-16〉J-Kフリップフロップ

(a) 記号

入力				出力	動作
R̄	C	J	K	Q	
0	×	×	×	0	リセット
1	↑	1	1	$\overline{Qn}$	反転
1	↑	1	0	1	セット
1	↑	0	1	0	リセット
1	↑	0	0	Qn	保持
1	↓	×	×	Qn	保持

(b) 真理値表

ディレイ回路

遅延回路．Dフリップフロップのクロック端子をシステム・クロックに直結すると，Dフリップフロップはたんにクロックの立ち上がりでD入力の値をストローブして出力するだけの働きしかできない．結果として入力の値が少し遅れて出力に伝達されるだけの回路となってしまう．

〈図5-17〉Dフリップフロップによる J-Kフリップフロップの回路

〈リスト5-2〉J-Kフリップフロップ(論理演算子による記述)

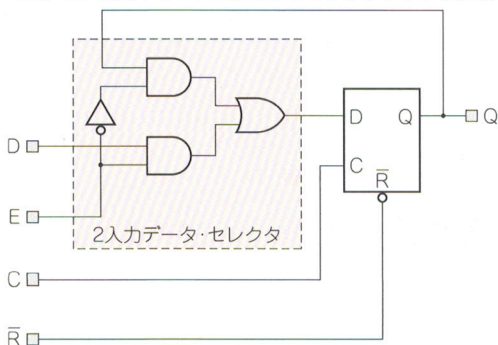
```
--
-- JK-F.F. (use logical equation)
--
library ieee;
use ieee.std_logic_1164.all;

entity jkf_A is
          port(   j  : in std_logic;
                  k  : in std_logic;
                  c  : in std_logic;
                  nr : in std_logic;
                  q  : out std_logic);
end jkf_A;

architecture rtl of jkf_A is

          signal bufQ : std_logic;

begin

          process(nr,c)
          begin

                  if (nr = '0') then
                          bufQ <= '0';
                  elsif (c'event and c = '1') then
                          bufQ <= (j and (not bufQ) ) or (k nor (not bufQ) );
                  end if;

          end process;

          q <= bufQ;

end rtl;
```

〈リスト5-3〉 J-Kフリップフロップ(if～then～else文による記述)

```
--
-- JK-F.F. (use if-then-else statement)
--
library ieee;
use ieee.std_logic_1164.all;

entity jkf_B is
          port(    j   : in std_logic;
                   k   : in std_logic;
                   c   : in std_logic;
                   nr  : in std_logic;
                   q   : out std_logic);
end jkf_B;

architecture rtl of jkf_B is

          signal bufQ : std_logic;

begin

          process(nr,c)
          begin

                  if (nr = '0') then
                          bufQ <= '0';
                  elsif (c'event and c = '1') then

                          if    (j = '1' and k = '1') then
                                  bufQ <= not bufQ;
                          elsif (j = '1' and k = '0') then
                                  bufQ <= '1';
                          elsif (j = '0' and k = '1') then
                                  bufQ <= '0';
                          end if;

                  end if;

          end process;

          q <= bufQ;

end rtl;
```

リップフロップは，データ入力(D)とクロック入力(C)をもっていますが，クロック入力がシテスム・クロックに直結されてしまうと，Dフリップフロップはシステム・クロックの立ち上がりでデータを取り込むだけの動作しか行えません．

Dフリップフロップは，同期設計を行おうとする場合においても，重要な基本的要素ですが，Dフリップフロップ自体は，クロックの立ち上がりの有無によっ

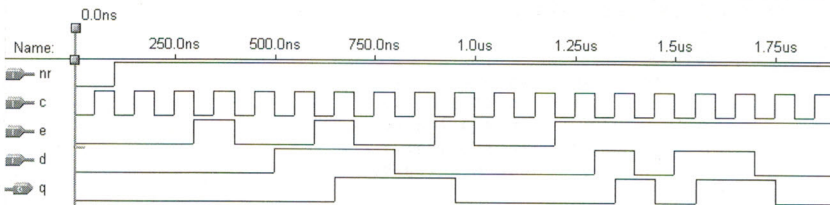
〈図5-18〉 同期イネーブル付きDフリップフロップの回路

〈図5-19〉 同期イネーブル付きDフリップフロップ

(a) 記号

入力				出力	動作
R̄	C	E	D	Q	
0	×	×	×	0	リセット
1	↑	1	1	1	データのロード
1	↑	1	0	0	
1	↑	0	×	Qn	保持
1	↓	×	×	Qn	

(b) 真理値表

〈リスト5-4〉同期イネーブル付きDフリップフロップ

```
--
-- D-F.F. with synchronous enable
--
library ieee;
use ieee.std_logic_1164.all;

entity dfe is
        port(   d  : in std_logic;
                e  : in std_logic;
                c  : in std_logic;
                nr : in std_logic;
                q  : out std_logic);
end dfe;

architecture rtl of dfe is
begin

        process(nr,c)
        begin

                if (nr = '0') then
                        q <= '0';
                elsif (c'event and c = '1') then
                        if (e = '1') then
                                q <= d;
                        end if;
                end if;

        end process;

end rtl;
```

てしかデータ・ロードのコントロールができない，言わば非同期設計用のフリップフロップなのです．

それでは，フリップフロップにクロックが入りっ放しの状態で，どうすればデータのロードをコントロールすることができるのでしょうか．もうすでに，クロック入力はシステム・クロックに直結されているし，非同期リセット入力はそのような制御の役には立ちません．結局，**データのラインに細工を施す**しか手はないようです．

データのラインに細工を施す
クロック端子にはシステム・クロックが直結され，リセット端子はデータ・ロードの制御には役立たないとなれば，データ(D)入力端子へ入力される値をコントロールするしか方法はない．

図5-18のように，Dフリップフロップと2入力データ・セレクタを組み合わせ，セレクタにより外部からのデータとDフリップフロップ自体の出力(Q)を切り替えて，Dフリップフロップの入力(D)にフィードバックすることにより，イネーブル入力(E)が'1'のときに外部データをロードし，E入力が'0'のときには自分自身の出力をリロードする(つまり，以前の状態を保持する)ことができます．

図5-19は同期イネーブル付きDフリップフロップです(**リスト5-4**)．

同期リセット/セット・フリップフロップの働き

同期リセット/セット・フリップフロップは同期リセット入力(SR)および同期セット入力(SS)をもったフリップフロップ(**フラグ**)です．同期リセット/セッ

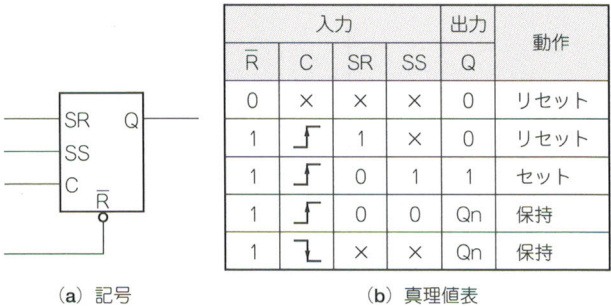
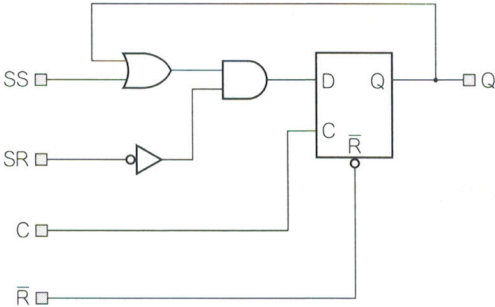
フラグ
flag(旗)の意．制御用のフリップフロップの出力を旗に見立ててフラグと呼ぶことがある．たとえばエラー発生時に'1'となるフリップフロップのことをエラー・フラグなどと呼んだりする．

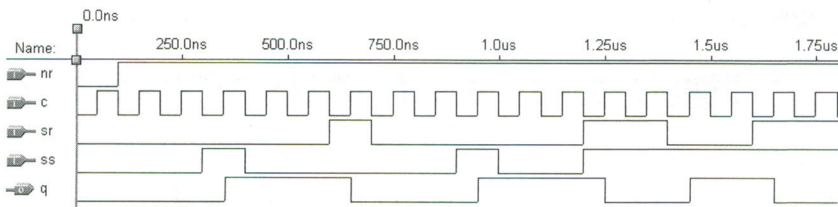
ト・フリップフロップでは二つの同期入力のうち同期リセット入力のほうが，優先順位が高くなります．また，初期設定のために，非同期リセット入力ももたせています(**図5-20**)．

Dフリップフロップを用いた同期リセット/セット・フリップフロップの回路を**図5-21**に示します(**リスト5-5**)．

〈図5-20〉同期リセット/セット・フリップフロップ

入力				出力	動作
$\overline{R}$	C	SR	SS	Q	
0	×	×	×	0	リセット
1	↑	1	×	0	リセット
1	↑	0	1	1	セット
1	↑	0	0	Qn	保持
1	↓	×	×	Qn	保持

(a) 記号　　(b) 真理値表

〈図5-21〉同期リセット/セット・フリップフロップの回路

〈リスト5-5〉同期リセット/セット・フリップフロップ

```
--
-- synchronous reset-set F.F.
--
library ieee;
use ieee.std_logic_1164.all;

entity rsf is
        port(   sr  : in std_logic;
                ss  : in std_logic;
                c   : in std_logic;
                nr  : in std_logic;
                q   : out std_logic);
end rsf;

architecture rtl of rsf is
begin

        process(nr,c)
        begin

                if (nr = '0') then
                        q <= '0';
                elsif (c'event and c = '1') then
                        if (sr = '1') then
                                q <= '0';
                        elsif (ss = '1') then
                                q <= '1';
                        end if;
                end if;

        end process;

end rtl;
```

同期セット/リセット・フリップフロップの働き

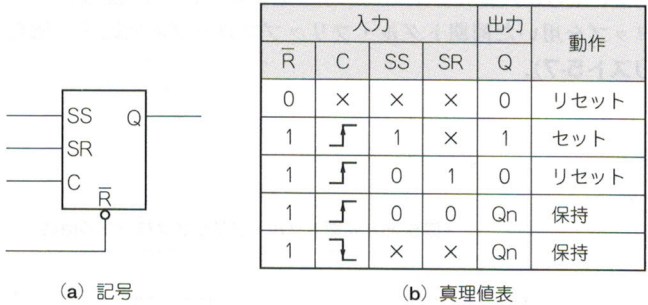
同期セット入力(SS)および同期リセット(SR)をもったフリップフロップ(フラグ)です．同期セット/リセット・フリップフロップでは，二つの同期入力のうち同期セット入力のほうが**優先順位が高く**なります．また，初期設定のために，非

優先順位が高く
たとえば，同期セット/リセット・フリップフロップの場合，同期リセット入力より，同期セット入力の優先順位が高くなる．同期セット入力と同期リセット入力が同時に'1'となった場合には，クロックの立ち上がりで出力がセットされることになる．

〈図5-22〉同期セット/リセット・フリップフロップ

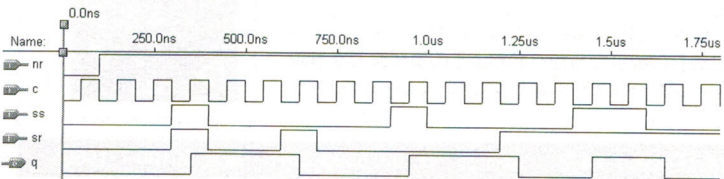
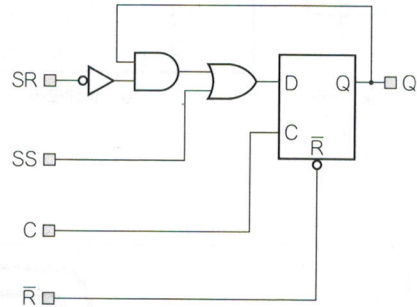
(a) 記号

入力				出力	動作
R̄	C	SS	SR	Q	
0	×	×	×	0	リセット
1	↑	1	×	1	セット
1	↑	0	1	0	リセット
1	↑	0	0	Qn	保持
1	↓	×	×	Qn	保持

(b) 真理値表

〈リスト5-6〉同期セット/リセット・フリップフロップ

```
--
-- synchronous set-reset F.F.
--
library ieee;
use ieee.std_logic_1164.all;

entity srf is
        port(   ss  : in std_logic;
                sr  : in std_logic;
                c   : in std_logic;
                nr  : in std_logic;
                q   : out std_logic);
end srf;

architecture rtl of srf is
begin

        process(nr,c)
        begin

                if (nr = '0') then
                        q <= '0';
                elsif (c'event and c = '1') then
                        if (ss = '1') then
                                q <= '1';
                        elsif (sr = '1') then
                                q <= '0';
                        end if;
                end if;

        end process;

end rtl;
```

〈図5-23〉同期セット/リセット・フリップフロップの回路

0.0ns
Name: 250.0ns 500.0ns 750.0ns 1.0us 1.25us 1.5us 1.75us
nr
c
ss
sr
q

同期リセット入力をもたせています(**図5-22**).

Dフリップフロップを用いた同期セット/リセット・フリップフロップの回路を，**図5-23**に示します(**リスト5-6**).

同期トグル・フリップフロップの働き

トグル動作の制御
同期設計をしようとして，Tフリップフロップのクロック入力(T入力)をシステム・クロックに直結すると，クロックの立ち上がりで出力が反転するだけの動作しかできなくなる．同期設計を行う際には，クロック入力とトグル動作を制御する入力を分離したフリップフロップが必要となる．

Tフリップフロップは，同期回路の設計用に使うことはできません．このため，**トグル動作の制御**を同期入力により行わせるようにしたものです．同期カウンタの基本回路となっているのが，このタイプのフリップフロップです(**図5-24**).

Dフリップフロップを用いた同期トグル・フリップフロップの回路を，**図5-25**に示します(**リスト5-7**).

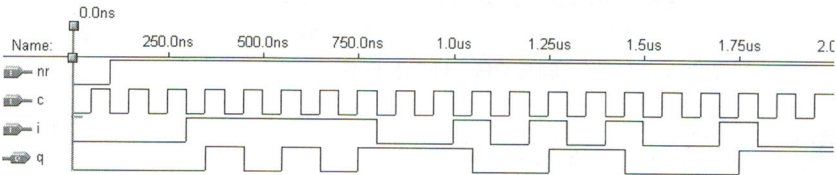
〈図5-24〉同期トグル・フリップフロップ

入力			出力	動作
R̄	C	I	Q	
0	×	×	0	リセット
1	↑	1	Q̄n	反転
1	↑	0	Qn	保持
1	↓	×	Qn	保持

(a) 記号　　(b) 真理値表

〈図5-25〉同期トグル・フリップフロップの回路

〈リスト5-7〉同期トグル・フリップフロップ

```
--
-- synchronous toggle F.F.
--
library ieee;
use ieee.std_logic_1164.all;

entity cf is
        port(   i   : in std_logic;
                c   : in std_logic;
                nr  : in std_logic;
                q   : out std_logic);
end cf;

architecture rtl of cf is

        signal bufQ : std_logic;

begin

        process(nr,c)
        begin

                if (nr = '0') then
                        bufQ <= '0';
                elsif (c'event and c = '1') then
                        if (i = '1') then
                                bufQ <= not bufQ;
                        end if;
                end if;

        end process;

        q <= bufQ;

end rtl;
```

同期イネーブル付きDレジスタの働き

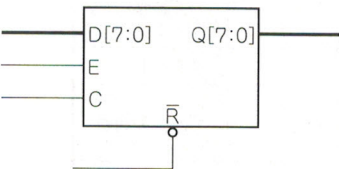
Dタイプ系のフリップフロップのデータまわりの信号をベクタ信号とすること

〈図5-26〉8ビット同期イネーブル付きDフリップフロップ

(a) 記号

入力				出力	動作
$\overline{R}$	C	E	D	Q	
0	×	×	×	0	リセット
1	↑	1	1	1	データのロード
1	↑	1	0	0	
1	↑	0	×	Qn	保持
1	↓	×	×	Qn	

(b) 真理値表

〈リスト5-8〉8ビット同期イネーブル付きDレジスタ

```
--
-- 8-bit D-reg. with synchronous enable
--
library ieee;
use ieee.std_logic_1164.all;

entity dfe8 is
        port(   d   : in std_logic_vector(7 downto 0);
                e   : in std_logic;
                c   : in std_logic;
                nr  : in std_logic;
                q   : out std_logic_vector(7 downto 0));
end dfe8;

architecture rtl of dfe8 is
begin

        process(nr,c)
        begin

                if (nr = '0') then
                        q <= (others => '0');
                elsif (c'event and c = '1') then
                        if (e = '1') then
                                q <= d;
                        end if;
                end if;

        end process;

end rtl;
```

多ビットのレジスタ
D型のフリップフロップを複数並列に並べると，複数ビットのデータを取り扱うレジスタを構成することができる．VHDLでは，ベクタ信号を使うことにより記述が簡単になる．

により容易に**多ビットのレジスタ**を得ることができます．ここでは，同期イネーブル付きDフリップフロップをベースとした，同期イネーブル付きDレジスタについて見てみましょう(**図5-26**，**リスト5-8**)．

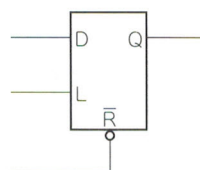
Dラッチの働き

レベル・トリガ・フリップフロップ
ラッチのこと．ラッチは，L入力のレベルによって，データ(D)入力の値をQ出力に通過させたり，その値を保持したりする．

最後になりましたが，**レベル・トリガ・フリップフロップ**の例として，Dラッチを取り上げます．Dラッチの記号と真理値表を，**図5-27**に示します．

Dラッチは，L入力が‘0’のときには，D入力の値がそのままQ出力へ伝達されます(データの通過)．そしてL入力が立ち上がると，その時点のD入力の値がQ出力に保持され，L入力が立ち下がるまで，その状態は変わりません(**リスト5-9**)．

〈図5-27〉Dラッチ

(a) 記号

入力			出力	動作
$\overline{R}$	L	D	Q	
0	×	×	0	リセット
1	0	1	1	データの通過
1	0	0	0	
1	1	×	Qn	保持

(b) 真理値表

〈リスト5-9〉Dラッチ

```
--
-- D-latch with asynchronous reset
--
library ieee;
use ieee.std_logic_1164.all;

entity latch is
        port(   d   : in std_logic;
                l   : in std_logic;
                nr  : in std_logic;
                q   : out std_logic);
end latch;

architecture rtl of latch is

begin

        process(nr,l,d)
        begin

                if (nr = '0') then
                        q <= '0';
                elsif (l = '0') then
                        q <= d;
                end if;

        end process;

end rtl;
```

正帰還をかけたバッファがメモリの始まり

ここにC-MOSインバータを2段接続した**バッファ**があるとします．**HC標準ロジック**なら，インバータ1段あたりの電圧ゲインは6倍前後なので，それを2段接続したバッファの場合は6×6で，30～40倍程度のゲインをもつことになります(**図5-28**)．

世の中では嘘つきは泥棒の始まりなどと言いますが，このバッファの入力を出力に接ないだものがメモリの始まりです．30倍前後のゲインをもったアンプに，100 %の正帰還がかかるので，一度各部の電位が‘0’と‘1’とに落ちついてしまうと，多少のノイズではデータが滑ったり転んだりすることはありません(**図5-29**)．

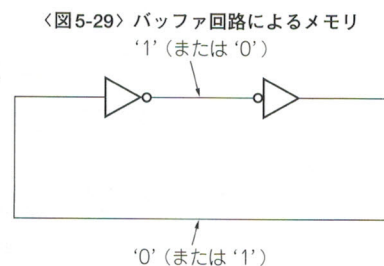
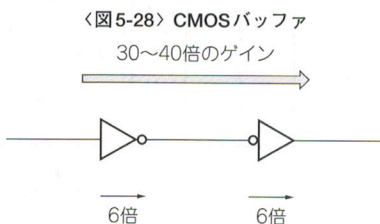
しかしこのメモリ，面白くも何ともありません．電源を立ち上げた直後に50％に近い確率で各部の電位が決まってしまうと，いったん電源を落として再度立ち上げるまではデータがびくともしないからです．

このようなことでは使いようがないので，昔のエライ人は考えたわけです．「正帰還のループを，閉じたり開いたりすることができるようにすれば，メモリ

バッファ
入力信号を反転せずに出力に伝達するロジック回路．

HC標準ロジック
3ミクロン前後のシリコン・ゲートC-MOSプロセスにより実現された，高速のC-MOS標準ロジックIC群．74LSシリーズTTLと互角の速度性能がある．それ以前の4000シリーズC-MOS標準ロジックの時代は，C-MOS ICと言えば消費電力は少ないものの低速であるというのが常識であったが，74HCシリーズ以後，その常識は崩れ，現在に至っている．

〈図5-28〉CMOSバッファ

〈図5-29〉バッファ回路によるメモリ

内のデータを変更したり，そのデータを保持することができるのではないか(**図5-30**)」と．

そこで取り出しましたるのは2個の**アナログSW**．これにより，バッファの入力をバッファの出力とデータ入力とに切り替えることにしたわけです．よくSWが切り替わる途中でデータが消えてしまわないのだろうかと心配する人がいますが，インバータには**入力容量**があるので短かい時間なら大丈夫です，と説明できるのがC-MOS ICの便利なところです(**図5-31**)．

こうしてでき上がったのが，通常はラッチと呼ばれるレベル・トリガ型のフリップフロップです(**図5-32**)．

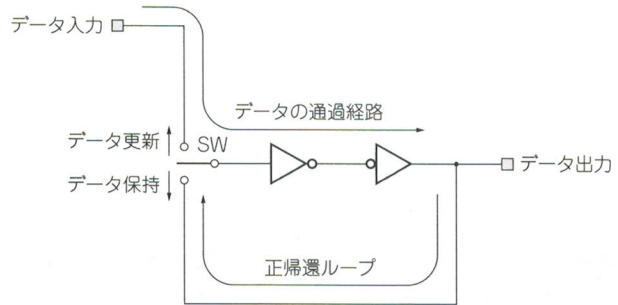
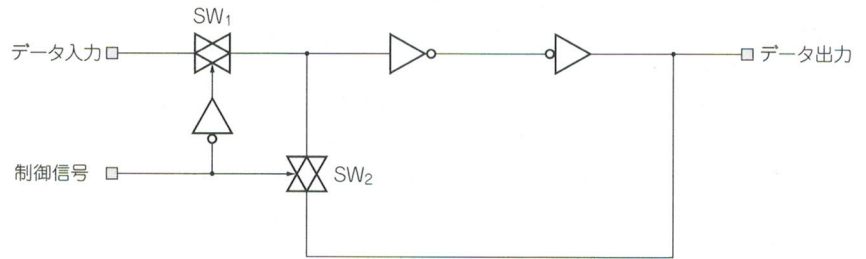
このラッチはラッチで使いようがあるのですが，カウンタなどを作る際には，制御信号のレベルにより入力データを通過させたり保持したりするのではなく，制御信号のエッジで取り込んだデータを次のエッジ変化まで保持してくれるほうが使いやすいということで，エッジ・トリガ型のフリップフロップ(通常，単に

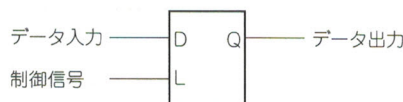
アナログSW

Pチャネル MOSトランジスタとNチャネル MOSトランジスタを向かい合わせに接続し，ゲート電圧を制御することにより，ディジタル信号ばかりではなくアナログ信号をも取り扱うことができる電子スイッチを構成することができる．

入力容量

C-MOS ICの入力端子はMOSトランジスタのゲートに接続されている．MOSトランジスタはその構造上，ゲート端子とドレイン/ソース端子間にpFオーダの入力容量をもつ．

〈図5-30〉データ書き換えが可能なメモリのイメージ

〈図5-31〉データが書き換え可能なメモリ(ラッチ)

備考）制御信号が‘0’のとき，SW_1がON，SW_2がOFF…スルー
制御信号が‘1’のとき，SW_1がOFF，SW_2がON…保持

〈図5-32〉レベル・トリガ型のフリップフロップ…ラッチ

注）制御信号のレベルが逆になっているラッチもある

(a) ラッチの記号

入力		出力	説　明
L	D	Q	
1	×	Qn	保持
0	1	1	スルー(データ通過)
0	0	0	

注）Qn：L入力が立ち上がる時点のQ出力の値

(b) ラッチの真理値表

フリップフロップと呼ばれる)の登場となります.

エッジ・トリガ型のフリップフロップは，ラッチを2段縦続接続した形で構成されます．制御信号は前段と後段で逆のレベルになるようにして，前段がデータを通過しているときには後段がデータを保持，前段がデータを保持しているときには後段がデータを通過するという動作を行わせます(**図5-33**).

このように，ラッチを2段構成にし，かならずどちらかが保持状態になるようにすれば，入力されたデータが筒抜けになることはなくなり，制御信号のエッジの変化点で入力データを取り込んだら，次のエッジの変化までそのデータを保持するという動作が可能になります.

この回路の場合，**データのストローブ・ポイント**は制御信号の立ち上がりエッジとなります．制御信号が‘0’のときには前段ラッチは入力データを出力に通過しているものの，後段のラッチが旧いデータを保持しているため，以前の制御信号の立ち上がり時に取り込まれたデータが出力されています.

データのストローブ・ポイント
この場合には，フリップフロップがデータを取り込む(ストローブ)タイミングのことを指す.

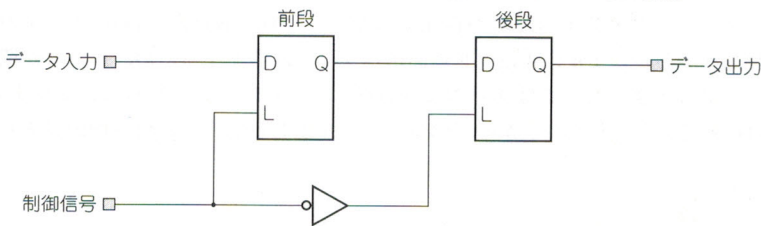
制御記号が‘0’から‘1’に変化した途端，前段ラッチはデータを保持し，後段ラッチは前段ラッチの出力をそのままフリップフロップの出力に伝達することになります．つまりこのタイミング(制御信号の立ち上がり)でフリップフロップはデータを取り込み，出力を更新するわけです.

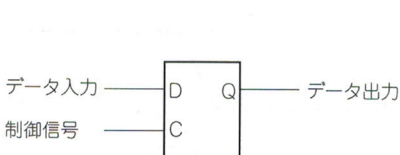
さらに，制御信号が‘1’から‘0’に変化すると，前段ラッチは次のデータの取り込みの準備に移りますが，それまで前段ラッチが出力していたデータを今度は後段ラッチが保持するため，この時点での出力変化は起こりません.

このように，データの取り込みはもっぱら前段のラッチが行い(マスタ)，後段のラッチはそれに付き従う(スレーブ)，そのようすから，**マスタ・スレーブ・フリップフロップ**などという呼び方をされることもあります(**図5-34**).

マスタ・スレーブ・フリップフロップ
エッジ・トリガ・フリップフロップの別称．回路を構成する二つのラッチが主従の関係にあることに由来する.

現在のディジタル回路設計においては，クロック(C)入力の立ち上がりエッジでデータを取り込むタイプのフリップフロップが主流となっています．しかし，タイミング的に非常にシビアな設計の場合などには，システム・クロックの立ち下がりでフリップフロップを動かすことも皆無とは言えません.

〈図5-33〉2段ラッチによるエッジ・トリガ型フリップフロップの構成

備考）制御信号が‘0’のときは，前段がスルー，後段が保持
制御信号が‘1’のときは，前段が保持，後段がスルー

〈図5-34〉Dフリップフロップ

(a)(D型)フリップフロップの記号

入力		出力	説明
C	D	Q	
↑	1	1	データの取り込み
↑	0	0	データの取り込み
↓	×	Qn	保持

(b)(D型)フリップフロップの真理値表

メタステーブルを考えなくてはいけないわけ

なぜVHDLの入門書でラッチやフリップフロップの話を延々と続けたかというと，VHDL設計者たる者，そのくらいの一般常識をもたなければならないから，ではなく，**メタステーブル**の説明をしたいからにほかなりません．

一般的に，メタステーブルの議論がなされているのは，もっぱらフリップフロップの場合に関してですが，メタステーブルは同じ記憶系のデバイスであるラッチやRAMなどでも起こり得ます．もし，これらの回路ではメタステーブルが起こらないのであれば，データのセットアップ時間やホールド時間の規定など必要ないということになります．

それでは，**セットアップ時間**や**ホールド時間**の規定が必要な理由について考えてみましょう(**図5-35**)．

セットアップ時間／ホールド時間とは，ラッチのL入力の立ち上がり，またはフリップフロップのクロック入力の立ち上がりを基準として，その前後のある期間，D入力のデータが変化せず安定していなければならないという規定です．ラッチのL入力またはフリップフロップのC入力の立ち上がりより，どれだけ遡った時点からデータが安定していなければならないかを定めるのがセットアップ時間で，立ち上がりの後どれだけデータの安定を保たねばならないかを定めたのがホールド時間です．

前項で述べたように，フリップフロップは**2段のラッチ**で構成されています．ラッチとフリップフロップの前段ラッチにおけるメタステーブルの発生プロセスは同一なので，以下ラッチの場合について解説します．

先に示した，セットアップ時間とホールド時間の規定は，ラッチのL入力の立ち上がり(フリップフロップの場合はクロック入力の立ち上がり)を中心に行われています．L入力の立ち上がりでは，ラッチ内のデータのループが開いた状態より閉じた状態へと遷移しています．これに前後する期間において，データの変化を禁止しているのがセットアップ時間とホールド時間です．

何が問題になるかというと，**図5-36**のA点よりB点までの**伝搬遅延**です．2段のインバータと若干の配線が存在するため，A点－B点間にはnsオーダの信号の遅れが存在します．A点－B点間はバッファ回路となるため，ループが開の場合，定常状態においてはA点およびB点のレベルはD入力と同じになります．この状態でL入力が立ち上がって帰還ループが閉じる場合には何ら問題はありません．

メタステーブル
フリップフロップが，出力状態が安定する以前の不安定な状態に陥った場合に，その状態をさす．

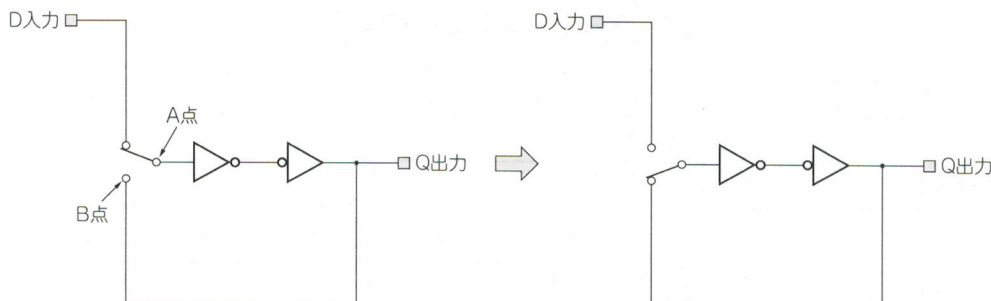
セットアップ時間
メタステーブルの発生を防ぐために，フリップフロップやラッチに課されているタイミング条件．制御信号(C入力やL入力)の変化以前に，どれだけの期間データ入力を変化させてはいけないかを規定している．

ホールド時間
メタステーブルの発生を防ぐために，フリップフロップやラッチに課されているタイミング条件．制御信号(C入力やL入力)の変化の後，どれだけの期間データ入力を変化させてはいけないかを規定している．

2段のラッチ
基本的な(エッジ・トリガ型)フリップフロップは2個のラッチを縦続接続することにより構成される．

伝搬遅延
論理回路の入力信号が変化した場合に，出力信号の変化までにかかる時間(遅れ)のこと．C-MOS回路においては，出力が抵抗成分をもち，入力は構造的に容量をもつため，基本的に伝搬遅延は*CR*時定数回路による遅れと考えることができる．

〈図5-35〉セットアップ時間とホールド時間

〈図5-36〉ラッチのループ(開→閉)

しかし，D入力が変化した直後は，伝搬遅延の関係で，短時間ですがA点のレベルとB点のレベルが違うという状態が生じます．この状態でL入力が立ち上がって帰還ループが閉じると，元々ループ・ゲインは30～40倍しかないので正帰還ループが安定するまでに時間がかかったり(出力の遷移に時間がかかる)，タイミングによってはループの中を信号の変化点がグルグル廻ったり(**発振状態**)するわけです．このような状態をメタステーブルと呼びます．

発振状態
フリップフロップにもよるが，クロックとデータ(D)入力の変化点のタイミングによっては，フリップフロップの出力が短時間の間，発振状態(これもメタステーブルの一種)となることがある．

メタステーブル対策

メタステーブルは，ラッチのL入力またはフリップフロップのクロック入力の立ち上がりの直後に起こりますが，C-MOS回路も特性的に完全な上下対称ではないので，ある程度の時間が経過すると，帰還ループ内の**正帰還**により回路の状態は安定していきます．この性質を利用して，メタステーブル対策を行うことができます．

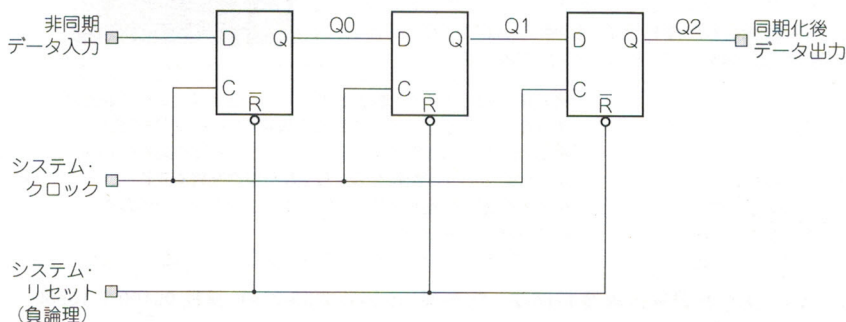
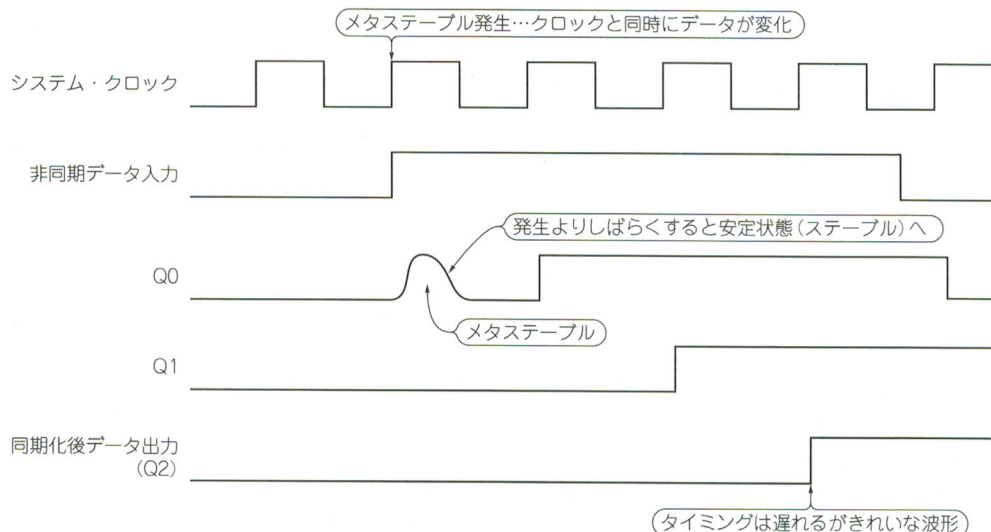
異なるシステムより信号が供給されるなどのために，システム・クロックの立ち上がりを基準としてその前後にデータが変化しない期間が確保できない場合は，シテスム・クロックにより駆動される2～3段のシフトレジスタを介して信号を取り込むことにより，メタステーブルの影響を取り除きます．

このような回路を**シンクロナイザ**(同期化器)と呼びます(**図3-37**，**図3-38**)．

メタステーブルは，クロックの立ち上がりに対して，セットアップ時間，ホールド時間を守らないでデータ入力が変化した場合，クロックの立ち上がりの直後よりフリップフロップの出力(**図3-37**，**図3-38**のQ0)に現れますが，しばらくすると落ちつくので，さらにフリップフロップを使ってサンプリングを繰り返す

正帰還
増幅器の出力を反転せずに入力に差し戻すこと．あまり帰還量を多くすると発振してしまう．0-V-2などの再生受信機(死語か？)では，増幅回路の正帰還量を調整することにより高感度を得ていた．

シンクロナイザ
同期化器．同期設計された回路において外部の非同期信号を取り込む際には，メタステーブルを避けるため，シンクロナイザを使ってその信号を同期化する必要がある．

〈図5-37〉シンクロナイザ

〈図5-38〉シンクロナイザの動作

確率

メタステーブルにより起こる症状は，フリップフロップのクロックの立ち上がりとデータの変化点の(時間的な)相互関係により決まる．データの変化がクロックの立ち上がりに対してまったくランダムであるとすると，メタステーブルにより起こる症状の酷さは確率に支配されることになる．

ことにより，メタステーブルの影響を排除することができます．

メタステーブルの持続時間は**確率**(クロックとデータの変化点の相互関係)により支配されるので，シフトレジスタ2段よりも3段のほうが，また，サンプリング周期が長いほうがメタステーブル除去の効果は高くなります．

第6章　回路をシステマティックに作ることを考えよう

階層設計と繰り返し表現

吉澤　清

階層設計の考え方

あらたまって階層設計などと言うと，何かたいへんなものなのではないかと身構えてしまいがちですが，何のことはありません．作った機能モジュールをほかの機能モジュールの上で部品(**コンポーネント**)として使おうというだけのことです．

コンポーネント
Component．構成要素の意．

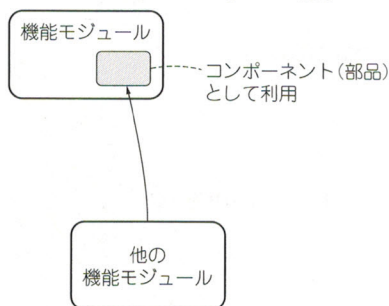
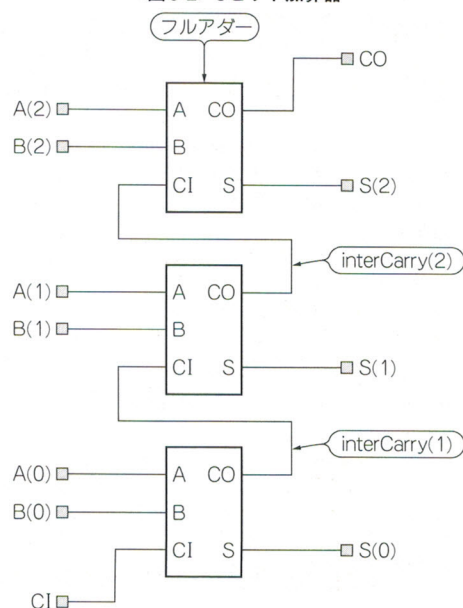
回路の規模が大きくなった場合に，回路をいくつかのブロックに分けて設計したり，頻繁に使う回路をブロック化しておいていろいろな回路で使うなどということは，回路図設計の場合においても日常茶飯事でした．同様のことをVHDLでもやろうというわけです．

ただ，いくつか決まり事があるので，以後そのあたりの説明をしていきます．

ロジックの記述ができて，フリップフロップが作れて，階層設計ができれば，これまで回路設計を行ってきた方は，VHDLの海を，水を得た魚のように泳ぎ廻ることができるはずです．

コンポーネントの利用

機能モジュールを記述する際には，ほかの機能モジュールをコンポーネント(部品)として使用することができます．VHDLにおいては，この方法により階層設計を実現します(**図6-1**)．

〈図6-1〉コンポーネントの利用

〈図6-2〉3ビット加算器

〈リスト6-1〉3ビット加算器のVHDLコード

```
--
-- full adder
--
library ieee;
use ieee.std_logic_1164.all;

entity fa is
          port(   a  : in std_logic;
                  b  : in std_logic;
                  ci : in std_logic;
                  co : out std_logic;
                  s  : out std_logic);
end fa;

architecture rtl of fa is
begin

          co <= (a and b) or ( (a or b) and ci);
          s   <= a xor b xor ci;

end rtl;

--
-- 3-bit adder
--
library ieee;
use ieee.std_logic_1164.all;

entity adder3 is
          port(   a  : in std_logic_vector(2 downto 0);
                  b  : in std_logic_vector(2 downto 0);
                  ci : in std_logic;
                  co : out std_logic;
                  s  : out std_logic_vector(2 downto 0) );
end adder3;

architecture rtl of adder3 is

          component fa
                  port(   a  : in std_logic;
                          b  : in std_logic;
                          ci : in std_logic;
                          co : out std_logic;
                          s  : out std_logic);
          end component;

          signal interCarry : std_logic_vector(2 downto 1);

begin

          mod2 : fa port map(
                  a  => a(2),
                  b  => b(2),
                  ci => interCarry(2),
                  co => co,
                  s  => s(2) );

          mod1 : fa port map(
                  a  => a(1),
                  b  => b(1),
                  ci => interCarry(1),
                  co => interCarry(2),
                  s  => s(1) );

          mod0 : fa port map(
                  a  => a(0),
                  b  => b(0),
                  ci => ci,
                  co => interCarry(1),
                  s  => s(0) );

end rtl;
```

注記：
- full adder（fa）の部分：コンポーネントとして使用する機能モジュールの記述
- 3-bit adder以降：3ビット加算器の記述
- component fa ～ end component：コンポーネント宣言
- signal interCarry：内部信号宣言
- mod2：コンポーネントの利用（2）
- mod1：コンポーネントの利用（1）
- mod0：コンポーネントの利用（0）

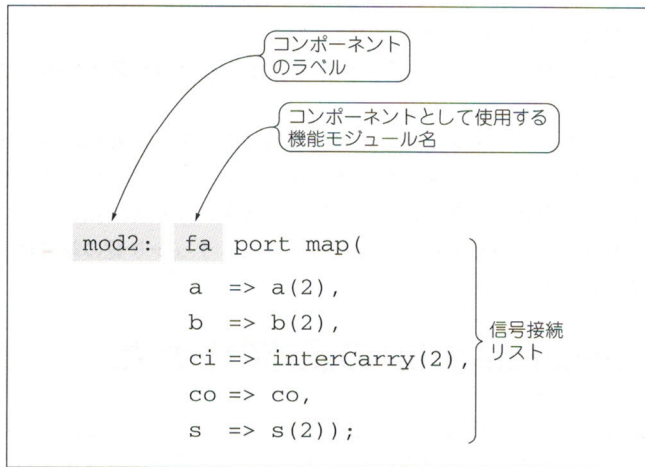
〈リスト6-2〉コンポーネント宣言とコンポーネントとして使われる機能モジュールのエンティティ

● コンポーネント宣言

```
component   fa
  port (a : in std_logic;
        b : in std_logic;
        ci: in std_logic;
        co: out std_logic;
        s : out std_logic);
  end  component  ;
```

●機能モジュールのエンティティ

```
entity  fa  is
  port (a : in std_logic;
        b : in std_logic;
        ci: in std_logic;
        co: out std_logic;
        s : out std_logic);
  end  fa  ;
```

〈リスト6-3〉port map文の書式(1)

〈リスト6-4〉port map文の書式(2)

```
mod2 : fa port map(a(2), b(2), interCarry(2), co, s(2) );
```

それでは，フルアダーの機能モジュールをコンポーネントとして利用して，3ビットの加算器を構成する場合を例にとって，コンポーネントの使い方を説明します(**図6-2，リスト6-1**)．

まず，コンポーネントとして使用する機能モジュールを，記述しようとする機能モジュールの上方に配置します．つまり，上位のモジュールになるほど，VHDLコードの下側に配置するわけです．一般的にコンパイラは，VHDLコードのいちばん最後に配置された機能モジュールを最上層と認識します．

ほかの機能モジュールをコンポーネントとして使用する場合には，まず**コンポーネント宣言**が必要となります．記述位置はアーキテクチャ部のarchitecture文とbeginの間，内部信号宣言用のsignal文の直前です．

コンポーネント宣言の書式は，コンポーネントとして使われる機能モジュールのエンティティ部の書式に酷似しています．したがって，機能モジュールのエンティティをコピーしてきて，一部を書き換えることにより，コンポーネント宣言とすることができます(**リスト6-2**)．

書き換えるポイントは3点．

▶1行目のentityをcomponentに変更

コンポーネント宣言

ほかの機能モジュールを部品(コンポーネント)として使うために，アーキテクチャ部の冒頭で行う宣言のこと．

▶1行目のisを削除

▶最終行のモジュール名(この場合はfa)をcomponentに変更

インデント
コードを記述する場合に，コードが見やすいように，入れ子となっている部分を右側にずらして書くようにすること．

なお，コンポーネント宣言はエンティティ部とくらべて1段**インデント**(文字下げ)されるので，文字の位置の調整を忘れてはいけません．

コンポーネント宣言を行うことにより，機能モジュールの中でコンポーネントの使用が可能になりました．実際にコンポーネントを使用するには，port map文を使います．port map文の書式を，**リスト6-3**に示します．

ラベル
コンポーネント同士を区別するために，コンポーネントを使用するためのport map文には，ラベルを付けることができる．

まず，コンポーネントの**ラベル**があります．今回のように，同じコンポーネントをいくつも使う場合には，それぞれを区別するのに便利です．続いて，コンポーネントとして使用する機能モジュール名(この場合はfa)を書きます．最後にport map(信号接続リスト); を書きます．信号の接続は，左辺にコンポーネント側の入出力信号名を，右辺にコンポーネントを使用する機能モジュール側の信号名を書き，その間を＝＞(等号と不等号による右向き矢印)で接ぎます．信号の接続の記述の間は**カンマ**(,)で区切ります．

カンマ
VHDLにおいては，構文の中に書かれる信号などを区切るために，カンマ(,)が使われる場合とコロン(;)が使われる場合がある．port map文の場合には，信号接続リストの区切りにカンマ(,)が使われる．

port map文の書き方には，もう一つあります．それは**リスト6-4**のようなものです．

このように，コンポーネント宣言における信号のリストの順に，コンポーネントを使用する機能モジュール側の信号名を並べることにより，信号の接続の表現が可能です．この書き方を使うと，記述自体は簡潔になりますが，信号と信号の対応が明示できないというマイナス点があります．うっかり者の筆者は，前出の記述法のほうを多用しています．

繰り返し表現(for ～ generate表現)の記述のしかた

階層設計は，設計効率を向上するためのよい方法です．しかし，コンポーネントの使用数が増えてくると，記述をするのがたいへんになってきます．

ベクタ表現
回路設計でいうバスのこと．一つの信号名で複数ビットのデータを表すことができる．

多ビットの演算回路において，コンポーネントに接続される信号名が，**ベクタ表現**によって一定の規則性をもっている場合には，for ～ generate文を使った繰り返し表現を使って，記述の量を減らすことができます．

それでは，5ビットの加算器の場合を例にとって，for ～ generate表現について見ていきましょう(**図6-3**)．

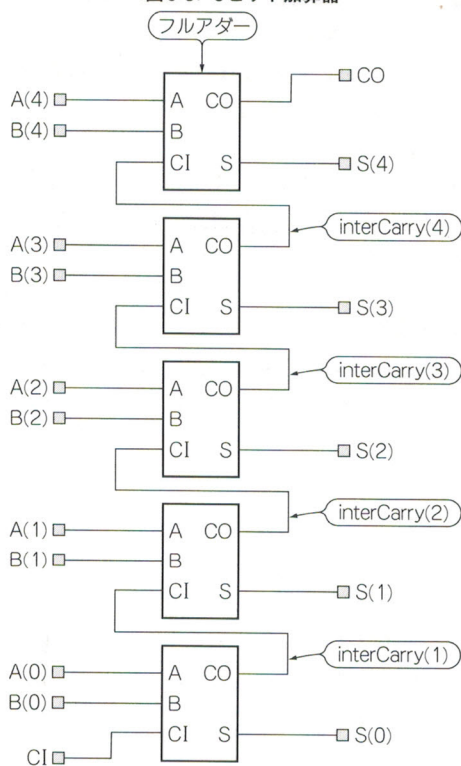
5ビット加算器をfor ～ generate表現を使って書くと，**リスト6-5**のようになります．

スキャン用変数
繰り返し表現を行う際に，繰り返し回数をカウントするために使われる特別な変数のこと．シーケンシャルな記述で使われるデータ(変数)とはまったく別のもの．回路上のデータとしてではなく，回路を記述する中でベクタ信号のビット指定などに使われることが多い．

図6-4に示すのがfor ～generate文の書式です．初めに，繰り返しループを区別するためのラベルがあります．for ～ in ～ generateは，**スキャン用変数**をスキャン範囲内で1ずつ更新しながら，end generate; までの処理を繰り返すという意味をもちます．

スキャン用変数は，繰り返し表現のためのループ・カウンタであり，ベクタ信号のビット指定にも用いられます．スキャン用変数は，間接的に回路の構造に影響を与えますが，スキャン用変数自体が回路上で実体化することはありません．また，スキャン用変数に関しては，使用する際に特別の宣言を行う必要はありません．for ～ in ～ generate文節のforとinとの間に名前を書くだけで使用することができます．

例では，スキャン範囲は3 downto 1となっています．この場合はfor ～ generate文は，スキャン変数(j)を3から1まで，一つずつ降順で変化させながらend generate; までの処理を繰り返します．もし，スキャン変数を昇順で変化さ

〈図6-3〉5ビット加算器

〈リスト6-5〉5ビット加算器をfor ～ generate文で書くと…

```
--
-- full adder
--
library ieee;
use ieee.std_logic_1164.all;

entity fa is
        port(   a   : in std_logic;
                b   : in std_logic;
                ci  : in std_logic;
                co  : out std_logic;
                s   : out std_logic);
end fa;

architecture rtl of fa is
begin

        co <= (a and b) or ( (a or b) and ci);
        s  <= a xor b xor ci;

end rtl;

--
-- 5-bit adder (use for-generate loop)
--
library ieee;
use ieee.std_logic_1164.all;

entity adder5f is
        port(   a   : in std_logic_vector(4 downto 0);
                b   : in std_logic_vector(4 downto 0);
                ci  : in std_logic;
```

〈リスト6-5〉5ビット加算器をfor ～ generate文で書くと…(つづき)

```
                    co  : out std_logic;
                    s   : out std_logic_vector(4 downto 0) );
end adder5f;

architecture rtl of adder5f is

          component fa
                    port(   a   : in std_logic;
                            b   : in std_logic;
                            ci  : in std_logic;
                            co  : out std_logic;
                            s   : out std_logic);
          end component;

          signal interCarry : std_logic_vector(4 downto 1);

begin

          mod4 : fa port map(
                    a  => a(4),
                    b  => b(4),
                    ci => interCarry(4),
                    co => co,
                    s  => s(4) );

          gen : for j in 3 downto 1 generate
                    mod1 : fa port map(
                            a   => a(j),
                            b   => b(j),
                            ci  => interCarry(j),
                            co  => interCarry(j+1),
                            s   => s(j) );
          end generate;

          mod0 : fa port map(
                    a  => a(0),
                    b  => b(0),
                    ci => ci,
                    co => interCarry(1),
                    s  => s(0) );

end rtl;
```

〈図6-4〉for ～ generate文の書式

せたい場合には，downtoでなくtoを使います．たとえば0から5まで，昇順で変化させる場合はスキャン範囲を0 to 5と記述します．とは言っても，for ～ generat文は，アーキテクチャ部のプロセス文の外(**コンカレントな記述**しか許されていない領域)でしか使えないため，スキャン方向が昇順だろうが降順だろうが，生成される回路に差異は生じません．今回は，記述の流れが，上位ビットから始まっているため，それに合わせてスキャン範囲を降順としているというわけです．

コンカレントな記述
同時並行的な記述．

〈リスト6-6〉5ビット加算器をfor ～ generate文を使わないで書くと…

```
--
-- full adder
--
library ieee;
use ieee.std_logic_1164.all;

entity fa is
        port(   a   : in std_logic;
                b   : in std_logic;
                ci  : in std_logic;
                co  : out std_logic;
                s   : out std_logic);
end fa;

architecture rtl of fa is
begin

        co <= (a and b) or ( (a or b) and ci);
        s   <= a xor b xor ci;

end rtl;

--
-- 5-bit adder (use for-generate loop)
--
library ieee;
use ieee.std_logic_1164.all;

entity adder5f_exp is
        port(   a   : in std_logic_vector(4 downto 0);
                b   : in std_logic_vector(4 downto 0);
                ci  : in std_logic;
                co  : out std_logic;
                s   : out std_logic_vector(4 downto 0) );
end adder5f_exp;

architecture rtl of adder5f_exp is

        component fa
                port(   a   : in std_logic;
                        b   : in std_logic;
                        ci  : in std_logic;
                        co  : out std_logic;
                        s   : out std_logic);
        end component;

        signal interCarry : std_logic_vector(4 downto 1);

begin

        mod4 : fa port map(
                a  => a(4),
                b  => b(4),
                ci => interCarry(4),
                co => co,
                s  => s(4) );

        mod3 : fa port map(
                a  => a(3),
                b  => b(3),
                ci => interCarry(3),
                co => interCarry(4),
                s  => s(3) );

        mod2 : fa port map(
                a  => a(2),
                b  => b(2),
                ci => interCarry(2),
                co => interCarry(3),
```

〈リスト6-6〉5ビット加算器をfor ～ generate文使わないで書くと…(つづき)

```
                  s   => s(2) );

          mod1  : fa port map(
                  a   => a(1),
                  b   => b(1),
                  ci  => interCarry(1),
                  co  => interCarry(2),
                  s   => s(1) );

          mod0  : fa port map(
                  a   => a(0),
                  b   => b(0),
                  ci  => ci,
                  co  => interCarry(1),
                  s   => s(0) );

end rtl;
```

以上のように，今回はスキャン変数jを3から1まで変化させながら，3個のfa(フルアダー)コンポーネントを発生(generate)しています．for ～ generat文を展開した場合のコードを**リスト6-6**に示しますので比較してみてください．

また，for ～ generate文は，論理演算子を用いた信号代入文などにも適用できます．たとえば，5ビット加算器は**リスト6-7**のように記述することが可能です．

リスト6-7のコードのfor～generat文を展開すると，**リスト6-8**のようになります．

〈リスト6-7〉5ビット加算器を論理演算子とfor ～ generate文で書くと…

```
--
-- 5-bit adder (use for-generate loop / logical equation)
--
library ieee;
use ieee.std_logic_1164.all;

entity adder5lf is
          port(   a   : in std_logic_vector(4 downto 0);
                  b   : in std_logic_vector(4 downto 0);
                  ci  : in std_logic;
                  co  : out std_logic;
                  s   : out std_logic_vector(4 downto 0) );
end adder5lf;

architecture rtl of adder5lf is

          signal interCarry : std_logic_vector(4 downto 1);

begin

          co   <= (a(4) and b(4) ) or ( (a(4) or b(4) ) and interCarry(4) );
          s(4) <= a(4) xor b(4) xor interCarry(4);

          gen : for j in 3 downto 1 generate
                  interCarry(j + 1) <= (a(j) and b(j) ) or ( (a(j) or b(j) ) and interCarry(j));
                  s(j)              <= a(j) xor b(j) xor interCarry(j);
          end generate;

          interCarry(1) <= (a(0) and b(0) ) or ( (a(0) or b(0) ) and ci);
          s(0)          <= a(0) xor b(0) xor ci;

end rtl;
```

〈リスト6-8〉リスト6-7のfor ～ generate文を展開すると…

```
--
-- 5-bit adder (use for-generate loop / logical equation)
--
library ieee;
use ieee.std_logic_1164.all;

entity adder5lf_exp is
          port(   a   : in std_logic_vector(4 downto 0);
                  b   : in std_logic_vector(4 downto 0);
                  ci  : in std_logic;
                  co  : out std_logic;
                  s   : out std_logic_vector(4 downto 0) );
end adder5lf_exp;

architecture rtl of adder5lf_exp is

          signal interCarry : std_logic_vector(4 downto 1);

begin

          co    <= (a(4) and b(4) ) or ( (a(4) or b(4) ) and interCarry(4) );
          s(4)  <= a(4) xor b(4) xor interCarry(4);

          interCarry(4) <= (a(3) and b(3) ) or ( (a(3) or b(3) ) and interCarry(3) );
          s(3)          <= a(3) xor b(3) xor interCarry(3);

          interCarry(3) <= (a(2) and b(2) ) or ( (a(2) or b(2) ) and interCarry(2) );
          s(2)          <= a(2) xor b(2) xor interCarry(2);

          interCarry(2) <= (a(1) and b(1) ) or ( (a(1) or b(1) ) and interCarry(1) );
          s(1)          <= a(1) xor b(1) xor interCarry(1);

          interCarry(1) <= (a(0) and b(0) ) or ( (a(0) or b(0) ) and ci);
          s(0)          <= a(0) xor b(0) xor ci;

end rtl;
```

レジスタ・フレームの考え方

ここで，同期設計による回路において，それぞれに**イニシャライズ用のリセット入力**とクロック入力をもつ，Dフリップフロップと8ビット・レジスタから成る**図6-5**のような回路があったとします．

このような回路を，Dフリップフロップと8ビット・レジスタそれぞれにプロセス文を書いて表現すると，**リスト6-9**のようになります．

しかし，この回路の場合，Dフリップフロップと8ビット・レジスタのクロック入力とリセット入力は，それぞれ共通のシステム・クロックとシステム・リセットに接続されているため，一つのプロセス文にまとめて記述することができます(**リスト6-10**)．

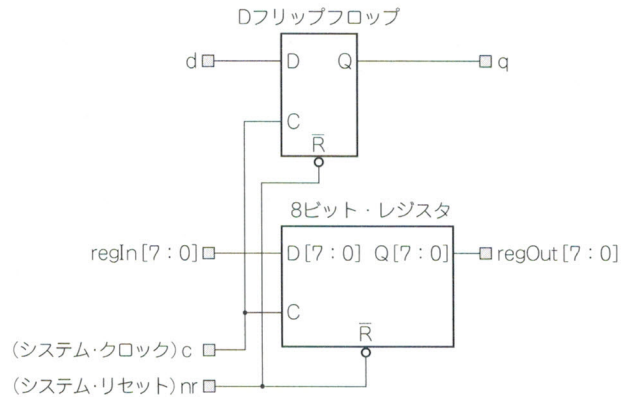
このように，記述を一つのプロセス文にまとめると，コード量を減らすことができます．通常，同期設計によるシステムでは，システム内のフリップフロップ，レジスタ，カウンタなどのクロック入力は，すべてシステム・クロックに接続され，また，内部回路の初期設定のためフリップフロップ，レジスタ，カウンタの**非同期セット/リセット/プリセット入力**などがシステム・リセットに接続されます．

このようなシステムにおいては，フリップフロップやレジスタ，カウンタを，一つのプロセス文にまとめることができます．

このプロセス文による機能をシンボルとして表すとしたら，**図6-6**のような

イニシャライズ用のリセット入力
回路を初期状態とするためのリセット入力．

非同期セット/リセット/プリセット入力
回路の初期データをall'1'や，all'0'，あるいは任意の値に設定するための非同期入力信号．

〈図6-5〉フリップフロップとレジスタからなる回路

ものになります．

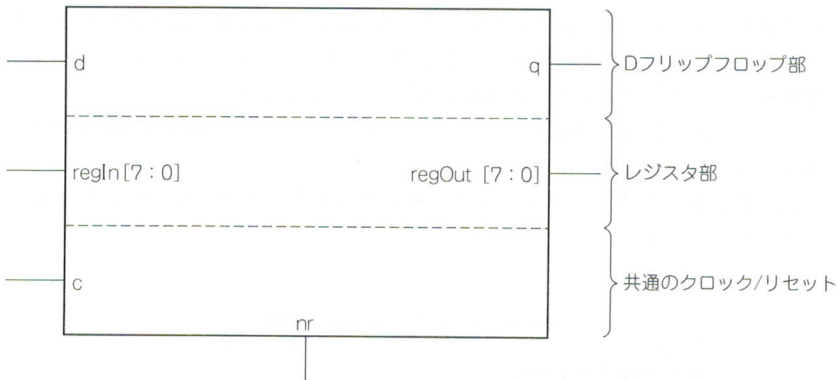
もちろん，一つの機能にはかならず一つのプロセス文を対応させるというような方針をおもちの方が，無理してこのように記述する必要はありません．なお，レジスタ・フレームというのは筆者の造語であり，一般的な用語ではありません．

〈リスト6-9〉Dフリップフロップと8ビット・レジスタそれぞれについてprocess文を書いた場合…

```
--
-- D-F.F. & 8-bit D-Reg.
--
library ieee;
use ieee.std_logic_1164.all;

entity regFF is
        port(   d       : in std_logic;
                regIn   : in std_logic_vector(7 downto 0);
                c       : in std_logic;
                nr      : in std_logic;
                q       : out std_logic;
                regOut  : out std_logic_vector(7 downto 0) );
end regFF;

architecture rtl of regFF is
begin

        process(nr,c)
        begin

                if (nr = '0') then
                        q <= '0';
                elsif (c'event and c = '1') then
                        q <= d;
                end if;

        end process;

        process(nr,c)
        begin

                if (nr = '0') then
                        regOut <= (others => '0');
                elsif (c'event and c = '1') then
                        regOut <= regIn;
                end if;

        end process;

end rtl;
```

〈リスト6-10〉リスト6-9を一つのprocess文にまとめて記述すると…

```
--
-- D-F.F. & 8-bit D-Reg. (reg. frame)
--
library ieee;
use ieee.std_logic_1164.all;

entity regFF_frame is
        port(   d       : in std_logic;
                regIn   : in std_logic_vector(7 downto 0);
                c       : in std_logic;
                nr      : in std_logic;
                q       : out std_logic;
                regOut  : out std_logic_vector(7 downto 0) );
end regFF_frame;

architecture rtl of regFF_frame is
begin

        process(nr,c)
        begin

                if (nr = '0') then
                        q       <= '0';
                        regOut  <= (others => '0');
                elsif (c'event and c = '1') then
                        q       <= d;
                        regOut  <= regIn;
                end if;

        end process;

end rtl;
```

〈図6-6〉フリップフロップとレジスタの複合体のイメージ

第7章　ポータビリティが高く，シーケンシャルな表現による記述ができる

ファンクションの使い方

吉澤　清

ファンクションとは

関数
　与えられた値に対して，ある規則に基づいた結果を返す機能をもつもの．

引き数
　関数に与える値のこと．

　ファンクションとは，言わずと知れた**関数**のことです．

　たとえば絶対値を返すabs(x)や三角関数のsin(x)などは，数学やコンピュータ言語上で使われている関数です．

　関数は，関数名(absとかsinとか)と，それに続くかっこ内の**引き数**により構成されます．たとえば，つぎのようにです．

```
abs(－0.5)
 ↑     ↑
関数名 引き数
```

　関数は引き数を元に計算を行い，計算結果を関数自体が返します．この関数が返す値のことを戻り値と呼びます．たとえば

```
y = abs(－0.5)
```

と書いたとすると，－0.5の絶対値が計算され，結果である0.5(戻り値)がyに代入されることになります．

　ここまでは数学の世界の話ですが，VHDLにおいても同様の形で利用可能な関数(function)を作り，それを使うことができます．

　関数を記述しようとするといろいろと憶えなければならないことが出てきますが，使うだけならたいへん簡単ですし，また便利でもあります．本章の後半に，複数のファンクションとその利用例があるので，まずは使ってみてください．そして，これ以外に自分が欲しいファンクションのイメージが沸いてきた時点で，本文に入っていただければと思います．

　VHDLにおけるファンクションは，**表7-1**のような特徴をもっていると考えることができます．

〈表7-1〉ファンクションのもつ特徴

- **ポータビリティが高い**
  …ライブラリ・パッケージの使用を宣言するだけで手軽に使える
- **利用可能な領域が広い**
  …プロセス文の内外を問わず，アーキテクチャ部の全域で利用可能
- **シーケンシャルな表現による記述ができる**
  …仮想的に時間的な分割設計を実現する手段

ライブラリ・パッケージの利用

　ファンクションは，機能モジュールとは別途に，ライブラリ・パッケージとして記述します．機能モジュール上でファンクションを使おうとする場合には，機

〈図7-1〉ファンクションの利用(increment8という名前のファンクションを使う場合)

ファンクションincrement8を含むライブラリ・パッケージ	ライブラリ・パッケージはVHDLコードの先頭部に置く

```
library ieee;
use ieee.std_logic_1164.all;
use work.ライブラリ名.all; …ライブラリの使用宣言

entity～
  ≀
end～;

architecture～
begin
  y<=increment8(a);  …ファンクションの利用
end～;
```

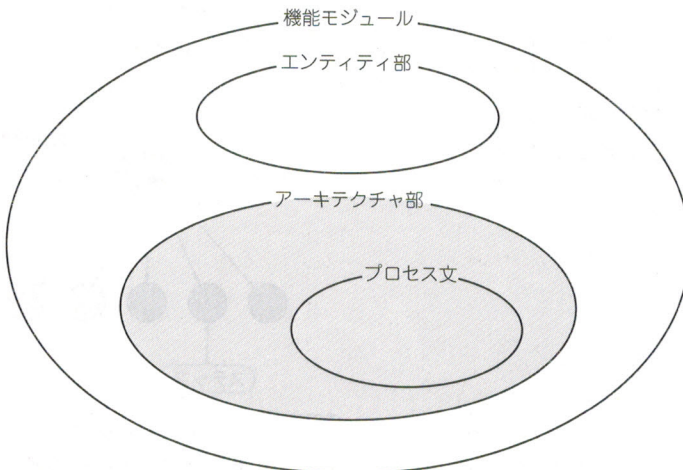
ファンクションを使用する機能モジュール

能モジュールの冒頭でuse文により，使いたいファンクションを含んでいるライブラリ・パッケージを使用する旨の**宣言**を行うだけですみます(**図7-1**)．

宣言
use文を使ってファンクション・ライブラリの使用宣言をすれば，その機能モジュール内(アーキテクチャ部)の全域で，そのライブラリのもつファンクションが使用可能になる．

ファンクションの利用可能な領域

VHDLにおいては，構文が使える領域が限定されていたりします．その点ファンクションは，機能モジュール上のアーキテクチャ部の全域(プロセス文の内か外かは問わない)で使うことができるので，非常に有用性が高いと言えます(**図7-2**)．

〈図7-2〉ファンクションが使用できる領域

シーケンシャルな表現と現実の回路との対応

コンカレントに動作する回路
シーケンシャルな記述を行った場合，それから合成されるのはコンカレントな回路である．シーケンシャルに動作をする回路を作る場合には，コンカレントな記述を使って，シーケンサやカウンタで制御される回路を記述してやらなくてはならない．なにかややこしい話だが，これが現実．

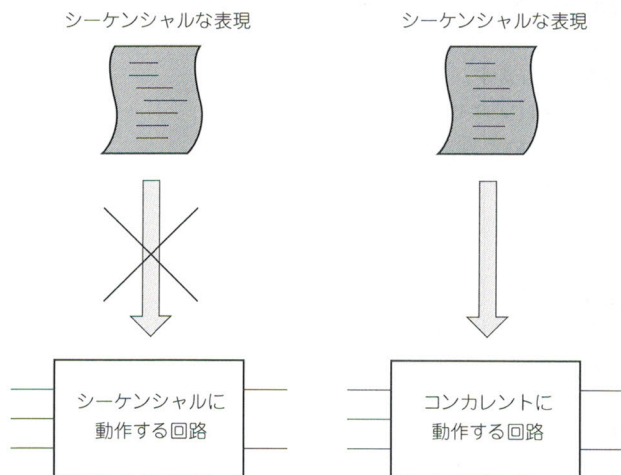
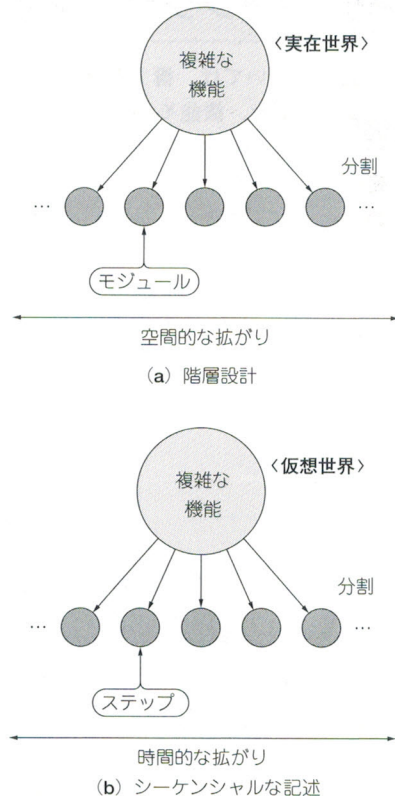
階層設計
VHDLにおいては，機能モジュールやファンクションを用いた階層設計により，システムを実現する(第6章を参照)．

表現をシーケンシャルに行うからと言って，回路合成ツールがそのコードを読んでシーケンシャルに働く回路を生成してくれるわけではありません．吐き出されてくるのは立派な(？)**コンカレントに動作する回路**なのです(**図7-3**)．

それではなぜ，わざわざシーケンシャルな表現などするのだろうという疑問が沸いてきます．大きな理由の一つは，処理の規則性に着目して，繰り返し表現(for〜loop文)を適用することにより，コード量を短く抑えることが可能なことです．コンカレントな表現においても繰り返し表現(第6章を参照)を使うことができますが，シーケンシャルな表現にくらべると信号の取り扱いが厳格であるために，適用することがかならずしも容易とは言えません．

そこでシーケンシャルな表現が登場したわけです．シーケンシャルな表現は，回路の動作を直接あらわすものではありません．かりに，回路の機能と同じ働きを，ステップに分割して表すとどう表現できるかを考えて，シーケンシャルに書いてみるわけです．複数のステップに分割することにより，全体として複雑な機能であったとしても，ステップごとの記述の複雑度は下げられるというわけです．

ある意味，「**階層設計**」を現実の下における空間的な分割設計であるとするならば，「シーケンシャルな記述」というものは，仮想的に時間的な分割設計を行うための方法であると言うことができます．結果として得られるのは，いづれの場合も現実のディジタル回路なのですが(**図7-4**)．

〈図7-3〉シーケンシャルな表現はコンカレントな回路を作るためにある

〈図7-4〉階層設計とシーケンシャルな記述

さて，回路合成ツールはシーケンシャルに表現されたコードをどのように取り扱うのでしょうか．回路合成ツールは，コードがシーケンシャルに実行された場合にどのような処理が行われるかを解析し，同様の結果を得ることができるような回路を生成するわけです．

シーケンシャルな表現で書かれたコードは，あくまでも仮想的な存在(つまり例え話のようなもの)なので，現実の回路の動作やコンカレントな表現で書かれたコードと，同列に取り扱おうとすると**混乱の原因**となります．

回路合成ツールにもよりますが，あまり複雑なシーケンシャル表現は，ツールの解析能力を超えたり，現実の回路で実現できなかったりすることがあるため，実際の設計に入る前に，いろいろと合成のテストをしてみることをお薦めします．

それでは，シーケンシャルな表現に必要な構文について解説していきましょう．

混乱の原因
シーケンシャルな表現による記述は，書かれた内容がそのまま実行されると考えると無理がある．シーケンシャルな記述を行おうとする場合には，コンカレントな記述を行うときとは頭を切り替えよう．

変数はシーケンシャルな表現で使われる仮想的なデータ

変数は，シーケンシャルな表現の中で使われる**仮想的なデータ**であり，ファンクションおよびプロセス文の中においてのみ使用することができます．

変数の取り扱いは，基本的に数学における数の取り扱いと同様であると考えて良いでしょう．変数に値が代入されると，次に新たな値が代入されるまでその値が保たれます．信号の場合と違って，変数に対しては幾度でも値を代入することができます．この上書きが可能であるという，ただそれだけのことが，シーケンシャルな表現に大きな融通性を与えています．

変数は，各ファンクションおよび各プロセス文の個々に属するローカルなデータであり，同じ変数名をあちこちで使ったとしても，たがいに干渉することはありません．また，ファンクション間やプロセス間で変数を用いて情報のやり取りをすることはできません．

仮想的なデータ
シーケンシャルな記述の中で使われる変数は，そのまま現実の回路の中において具現化されるとは限らない．変数は，仮想的に回路の動作をシーケンシャルに分割して記述するための，仮想的なデータである．

変数の宣言

変数を使用しようとする場合には，あらかじめ宣言を行わねばなりません．変数の宣言は，次のような書式で行います．

variable　変数名：型指定;

記述位置はファンクションの場合には，ファンクション本体の記述の中のfunction行とbeginの間であり(**リスト7-1**)，プロセス文の場合は，process(センシティビティ・リスト)とbeginの間です(**リスト7-2**)．

〈リスト7-1〉ファンクションにおける変数宣言の記述位置

```
package body ライブラリ名 is
    function ファンクションの仕様 is
         variable 変数名 : 型指定; ←———— 変数の記述位置
    begin
     ∫
    end ファンクション名;
end ライブラリ名;
```

〈リスト7-2〉プロセス文における変数宣言の記述位置

```
process   (センシティビティ・リスト)
          variable 変数名 : 型指定;           変数の記述位置
begin
     ∫                          ←
end;
```

変数代入文の書式

上書き
一度代入された値に，重ねてさらに代入を行うこと．信号の場合は上書きをすることはできない．

変数に値を代入しようとする場合には，変数代入文(：＝で表される)を用います．変数代入文の書式を**図7-5**に示します．

変数代入文の取り扱いは，信号代入文のそれに準じます．**上書き**(多重の代入)が可能である点だけが，信号代入文とは異なります．

〈図7-5〉**変数代入文の書式**(ファンクション上での記述)

（注）プロセス文の中では，変数代入文の右辺に，さらに次のものが書ける．
- ▶信号
- ▶信号を用いた演算式
- ▶ファンクション

for～loop文の書式

for～loop文は，ファンクションおよびプロセス文の中で使われる，繰り返し表現を行うための構文です．プロセス文の外では記述することができません．

for～loop文の書式を**図7-6**，**図7-7**に示します．

ループ変数
繰り返しループの回数を数えるための特別な変数のこと．

アップ・スキャンは，**ループ変数**が小さい値から大きい値へと変化する場合を指し，その場合は開始値＜終了値という関係にあります．ダウン・スキャンはそれとは逆に，ループ変数が大きな値から小さな値へと変化する場合を指し，開始値＞終了値の関係にあります．二つの書式で異なっているのは，開始値と終了値の間に書かれるキーワードがtoであるかdowntoであるかだけです．

それでは書式の説明に移ります．まずラベル(オプション)があります．続いて，

〈図7-6〉**for～loop文の書式**(アップ・スキャン時)

```
ラベル : for ループ変数 in 開始値 to 終了値 loop
           処理
end loop;
```

〈図7-7〉**for～loop文の書式**(ダウン・スキャン時)

```
ラベル : for ループ変数 in 開始値 downto 終了値 loop
           処理
end loop;
```

キーワードのforがありループ変数が続きます．ループ変数は基本的には回路上で取り扱われるデータ(信号，定数，変数)とはまったく別物であり，for～loop文の繰り返し制御用のループ・カウンタとして使われ，また**ベクタ信号のビット指定**および範囲指定などのための値として使うことができます．VHDLは厳格な言語で，データを使用しようとする場合には，通常事前に定義をしなければなりませんが，このループ変数だけは例外のようで，この位置(forとinの間)に名前を書けばfor～loop文内で通用します．教科書では，プログラミングの影響を受けてIという名前が好まれるようですが，数字の 1 や小文字の l との混同を防ぐため，本書では j を使っています．

開始値はループ変数の初期値であり，for～loop文は開始値より終了値までループ変数を1ずつ変化させながら，loopとend loop; の間に書かれた処理を繰り返します．開始値<終了値の場合はループ変数は，ループが繰り返すたびに＋1され(アップ・スキャン)，開始値>終了値の場合はループ変数は，ループが繰り返すたびに－1され(ダウン・スキャン)ます．

ベクタ信号のビット指定
ベクタ信号を定義する際には，範囲の指定が必要になることは第3章で解説した．ベクタ信号を構成する各ビット・データには，この範囲の指定に基づいてインデックス用の番号が振られている．ベクタ信号のビット指定を行う場合には，このインデックス番号を使用する．

'high，'lowアトリビュート

第5章の'eventアトリビュートの項でも述べたように，アトリビュートというのは**データに付随する情報**のことです．くだけた言い方をするならば，そいつがどんな奴であるかを表す情報ということができます．Windowsで出てくるプロパティみたいなものなのです．

'high，'lowアトリビュートは，データの範囲に関する情報を取り出すためのものであって，ベクタ・データ(ベクタ信号とベクタ変数)に対して適用されます．'highアトリビュートを使うと，データ範囲の高いほうの値を，'lowアトリビュートを使うと，データ範囲の低いほうの値を取り出すことができます．ただし，値とは言っても，基本的には回路上で取り扱うデータとなるわけではありません．

たとえば，std_logic_vector(7 downto 0)という型をもつベクタ信号aがあったとした場合，そのアトリビュートa'highは7，a'lowは0となります．さて，このようなものをどのように使うのかというと，**データ範囲のリンク**に使用するわけです．たとえば，さきほどのベクタ信号aと同じデータ範囲をもつベクタ変数bを定義しようとした場合，

```
variable b : std_logic_vector(7 downto 0);
```

と宣言することができますが，これを

```
variable b : std_logic_vector(a'high downto a'low);
```

という形で宣言することもできるというわけです．このような記述を行うことによるメリットは，ベクタ変数bのデータ範囲がベクタ信号aのデータ範囲と同じになることです．つまり信号aのデータ範囲を書きかえるだけで，変数bのデータ範囲も一緒に変わってくれるというわけです．

また，'high，'lowアトリビュートはfor～loop文における繰り返し範囲の指定にも使うことができます．たとえば，前出の信号a(std_logic_vector(7 downto 0)型)のLSBから順にMSBまで，1ビットごとに処理していきたい場合，

```
for j in a'low to a'high loop
    処理
end loop;
```

データに付随する情報
'highと'lowアトリビュートはデータのもつ範囲情報を抽出するための機能をもっている．

データ範囲のリンク
あるデータの範囲を，ほかのデータのデータ範囲に合わせること．

という形でfor～loop文を書き，処理の中でa(j)などの表現をすれば，ループが繰り返されるごとに，ベクタ信号aの各ビットを参照することができます．同様のスキャンは，

```
for j in 0 to 7 loop
    処理
end loop;
```

と書くこともできますが，この書き方では，取り扱う信号のデータ範囲が変化した場合にはそれに合わせて，for～loop文の繰り返し範囲も書き換えなければなりません．その点，アトリビュートを使うと，繰り返し範囲を，取り扱うデータの範囲から自動的に引用することができるので，データの範囲を変更する場合のコードの書き換えの手間が省けます．

また，信号aのMSBから順にLSBまで1ビットごとに処理していきたい場合には，

```
for j in a'high downto a'low loop
    処理
end loop;
```

というように書きます．もしMSBの1ビット下からスキャンを始めたいような場合には，

```
for j in a'high-1 downto a'low loop
    処理
end loop;
```

というように書くことも可能です．アトリビュートの値に対しては，整数演算が可能というわけです．

また，ループ変数(この場合はj)が示すビットより一つ上のビットを取り扱いたいというような場合には，a(j＋1)などという表現を用いることもできます．ループ変数の値に関しても整数演算の適用が可能です．

アトリビュートは便利なのですが，コードを書く場合に多少繁雑なので，始めのうちは具体的に数値で範囲指定をし，しかるのちにアトリビュートを使った書き方に移行したほうが，入門は楽であるかもしれません．

return文の書式

return文は，ファンクション内部で変数を用いて計算された値を，関数の戻り値とするための構文です．

たとえばファンクション内での演算結果が，最終的にtempという名前の変数に入る場合，**ファンクション内の記述のラスト**で，

```
return(temp);
```

と書くことにより，tempの値がそのファンクションの戻り値となります．

ファンクション内の記述のラスト
この場合，変数tempの値はファンクション内における処理の結果，ファンクションの最後でその値が確定することになる．

Max＋plus Ⅱ上ではファンクションはこう使う

VHDLにおいてシーケンシャルな表現(ファンクションの記述)を行おうとする場合，Max＋plus Ⅱには若干の制限があるようです．これらは将来解除される方向にあるようですが，本書ではこれらの制限の範囲内で記述をしていきます．その制限とは，次のようなものです．

Max+plusII
アルテラ社のCPLD開発環境．回路合成ツールや合成後シミュレータ，ほかの機能をもつ総合開発環境である．最近ではweb版のものでも，VHDLコードを取り扱うことができるようになっている．

▶ Max＋plus Ⅱ上では，for～loop文の多重ループ表現は許されない
　つまり，for～loop文の中でまたさらにfor～loop文を書くことはできません．
▶ ファンクション呼び出しからファンクション本体に，引き数のデータ範囲の情報が伝わらない
　ファンクションを作る際，**可変長の引き数**に対応することができません．
▶ ファンクションを呼ぶ際に，引き数のデータ範囲がファンクションの引き数のデータ範囲の定義と一致しなければならない
　たとえば，ファンクションの記述において引き数の定義がstd_logic_vector (7 downto 0)である場合，ファンクションを呼ぶ際の引き数となるデータもstd_logic_vector(7 downto 0)という型でなければなりません．std_logic_vector(8 downto 1)などという型をもつデータは，引き数としては使えないわけです．

可変長の引き数
VHDLの仕様では，関数に与えられた引き数のデータ範囲の情報を，関数の側でアトリビュートにより参照できることになっている．このため，引き数のデータ範囲が変化したとしても，対応できる筈なのだが….

このため，本書では次のような方針のもとで，ファンクションの記述を行っていきます．
▶ for～loop文は単一ループで運用する
▶ ファンクションは，固定長の引き数に対応させる．複数のデータ長の引き数への対応に関しては，データ長ごとのファンクションを作成することにより行う．
▶ ファンクションの引き数の定義と，実際に与えられる引き数のデータ範囲がずれている場合には，いったん，ファンクション側の引き数の定義と同じデータ範囲をもつデータに代入し，これをファンクションに渡すようにする．

5ビットの引数の値に1を加えたものを戻り値とする関数increment5f

それでは，実際にファンクションを含んだライブラリ・パッケージを作ってみましょう．ここでは，5ビットの引き数の値に1を加えたものを戻り値とするincrement5fというファンクションを取り上げます．increment5fは，次の項で登場するincrement5と同じ機能をもちます．ただし，記述の中に'high，'lowアトリビュートを含んでいません．両者の違いを，これら二つのアトリビュートの使い方の参考としてください(**リスト7-3**)．

まず，お決まりのstd_logic，std_logic_vector型を使うためのライブラリ使用宣言があります．

続いて，ファンクションの宣言部があります．この部分の書式は，

```
package ライブラリ名 is
    ファンクション宣言
end ライブラリ名;
```

〈リスト7-3〉increment5fファンクションを含むライブラリ・パッケージlibIncFの記述

```
--
-- increment library (not use 'high , 'low attributes)
--
library ieee;
use ieee.std_logic_1164.all;
package libIncF is
   function increment5f(sorce : std_logic_vector(4 downto 0)) return std_logic_vector;
end libIncF;

package body libIncF is
   function increment5f(sorce : std_logic_vector(4 downto 0)) return std_logic_vector is
         variable andAll : std_logic;
         variable temp    : std_logic_vector(4 downto 0);
   begin
         andAll := '1';
         for j in 0 to 4 loop
             temp(j) := sorce(j) xor andAll;
             andAll := andAll and sorce(j);
         end loop;
         return(temp);
   end increment5f;
end libIncF;
```


〈図7-8〉ファンクション宣言の構造

というものです．ファンクション宣言は**図7-8**のような構造をもっています．

つぎに，ファンクション本体の記述部があります．この部分の書式は，

```
package body ライブラリ名 is
    ファンクション本体の記述
end ライブラリ名;
```

というものです．

ファンクション本体の記述はつぎのような書式で行われます．

```
function 関数名(引き数の名前と型指定) return 戻り値の型指定 is
    変数の宣言
begin
    実際の処理の記述
end 関数名;
```

なお，1行目の記述は**図7-8**の書式の末尾のセミコロンを除き，そのかわりにisを付け加えたものとなります．

5ビット・インクリメンタincrement5fファンクションの中身(アルゴリズム)

シーケンシャルな表現だからプログラムと同じだ，だから難しいという論もあるようですが，数千行のコードならともかく，本書で取り上げているシーケンシャルな表現のほとんどはたかだか数行しかありません．たった数行のコード，考え方さえわかってしまえば，回路設計者にとっては取るに足りるものではありません．

シーケンシャルな表現とは，通常われわれが回路で実現している**同時並行的な処理**をいくつかのステップにわけて，1ステップずつ実行していくような書き方です．たとえば，非常に簡単な例で，5ビットの変数a(:std_logic_vector(4 downto 0))のすべてのビットに1を代入しようとする場合，シーケンシャルに5ステップで(各ビットごとに)処理を行おうとすると，つぎのようなコードになります．

```
a(0)  :=  '1'  ;
a(1)  :=  '1'  ;
a(2)  :=  '1'  ;
a(3)  :=  '1'  ;
a(4)  :=  '1'  ;
```

シーケンシャルに書かれたコード(変数代入文を使って書かれたコード)は，上より順に実行されるものと考える(仮想する)ものとされているので，このコードの解釈は，まず変数aのビット0に1を代入し，つぎにビット1に1を代入，…といったものになります．しかし，このコードを回路合成した結果はかならずしも出力が順番に立ち上がっていくという回路にはなりません．なぜなら，回路合成ツールが**シーケンシャルな表現をコンカレントに働く実回路に変換する**からです．つまり，回路合成ツールは，シーケンシャルな表現をシーケンシャルに実行した場合に得られる最終結果と同じ結果が得られるような回路を吐き出すわけです．

シーケンシャルな表現を行う場合に，上述のコードのように1ビットずつ丁寧に記述してもよいのですが，for ～ loop文による繰り返し表現を用いると，処理が多くのビットにわたる場合，以下のようにコードが簡潔なものとなります．

```
for j in 0 to 4 loop
  a(j) :='1';
end loop;
```

それでは，本題の**インクリメンタ**を設計してみましょう．5ビット・インクリメンタの回路を**図7-9**に示します．

この回路を，

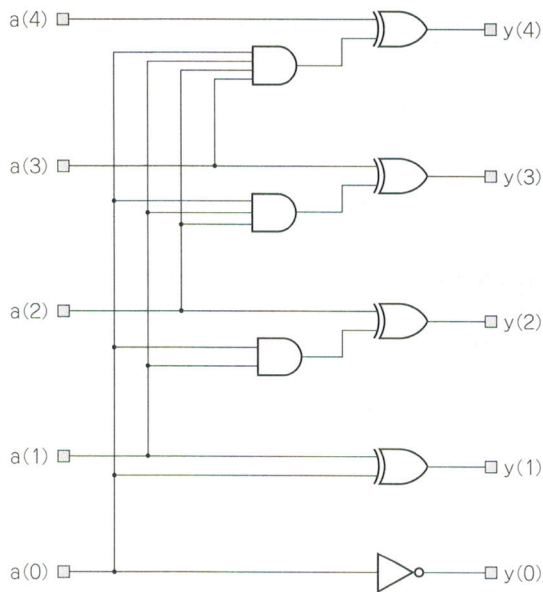
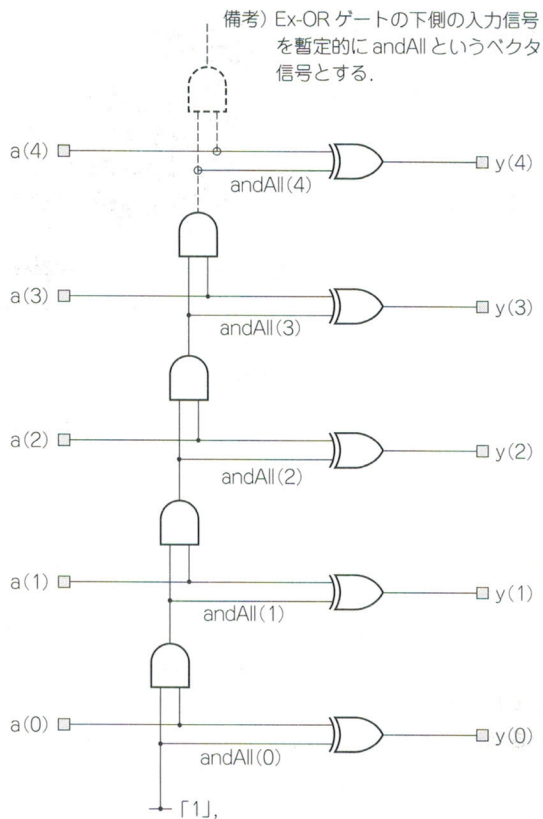
```
y(4)  := a(4)  xor  (a(3)  and  a(2)  and  a(1)  and  a(0));
y(3)  := a(3)  xor  (a(2)  and  a(1)  and  a(0));
y(2)  := a(2)  xor  (a(1)  and  a(0));
y(1)  := a(1)  xor  a(0);
y(0)  := not  a(0);
```

と書いたとしても，これではコンカレントな記述とまったく同じで，シーケンシャルな記述としてのメリットはありません．シーケンシャルな記述のメリットである繰り返し記述を適用するためには，各ビットの処理が一定の規則に則った形に変形してやらなくてはなりません．

同時並行的な処理
実際の回路の動作は同時並行的なものである．

シーケンシャルな表現をコンカレントに働く回路に変換する
VHDLにおけるシーケンシャルな表現は，コンカレントに動作する回路を記述するための手段であり，シーケンシャルに動作する回路を表現するためのものではない．回路合成ツールは，シーケンシャルな表現で書かれたVHDLの記述をコンカレントに動作する回路に変換する．

インクリメンタ
＋1演算を行う回路．

〈図7-9〉5ビット・インクリメンタの回路

〈図7-10〉書き換え後の5ビット・インクリメンタ

このような考え方に基づいて，インクリメンタの回路を書き換えると，**図7-10**のようになります．

このように書き直すことにより，実回路として望ましいかどうかは別として，各ビットごとの処理がまったく同じ形になります．これをシーケンシャルな表現を使って記述すると，以下のようになります．なお，この回路においては，上位側のビット処理は下位側のビット処理結果に依存するため，**コード**を下位のビットより順に記述していきます．

コード
VHDLコードのこと．

```
andAll(0)  := '1'; ..............................初期設定
y(0)  := a(0)  xor  andAll(0);                   ┐
andAll(1)  := andAll(0)  and  a(0);              ┘ビット0の処理
y(1)  := a(1)  xor  andAll(1);                   ┐
andAll(2)  := andAll(1)  and  a(1);              ┘ビット1の処理
y(2)  := a(2)  xor  andAll(2);                   ┐
andAll(3)  := andAll(2)  and  a(2);              ┘ビット2の処理
y(3)  := a(3)  xor  andAll(3);                   ┐
andAll(4)  := andAll(3)  and  a(3);              ┘ビット3の処理
y(4)  := a(4)  xor  andAll(4);
{andAll(5)  :=  andAll(4)  and  a(4);}             ビット4の処理
```

各ビットの処理は同じ形なので，for～loop文による繰り返し表現を使うと，次のように記述することができます．

```
andAll(0) := '1';
for j in 0 to 4 loop
    y(j) := a(j) xor andAll(j);
    andAll(j+1) := andAll(j) and a(j);
end loop;
```

また，変数はシーケンシャルに処理を行っていく過程において，上書きをすることが許されているので，とくに複数の値が必要になる場合(入力されたり，出力されるなど)を除けば，変数を使い回すことにより使用する変数の数を減らすことができます．たとえば，前出のコードにおいては，andAllという変数をベクタとしていますが，これを**非ベクタ信号**として使い回してもよいわけです．

```
andAll := '1';
for j in 0 to 4 loop
    y(j) := a(j) xor andAll;
    andAll := andAll and a(j);
end loop;
```

インクリメンタの処理の規則性は，データのビット数が増えても変わらないため，上記のコードにおいてループの繰り返し回数を増やすことにより，容易にビット長の長いインクリメンタを実現することができます．

この後，入力であるaを引き数に合わせてsorceという名前に，出力であるyを**一時的なデータ保持用の変数**tempに変更し，コード末尾でreturn(temp); としてtempの値を関数の戻り値としてアサインすれば，ファンクションの記述のできあがりです．

非ベクタ信号
ベクタ信号に対して，1ビットの信号（std_logic型）のことを言っている．

一時的なデータ保持用の変数
ファンクションにおける処理結果は，いったんデータ保持用の変数に格納されreturn文に渡される．

シーケンシャルな表現をするときには

以上，述べてきたように，シーケンシャルな表現で記述を行おうとするときには，回路図で回路を設計していたときとはちょっと頭を切り換えなくてはなりません．シーケンシャルな表現をしようとするときには，以下の点に気を付けましょう．

●とにかく，ステップに分けて処理した結果が必要な値になればよい

回路屋の癖で，どうしても細かいところにこだわりたくなるのはわかりますが，シーケンシャルな表現においては，まさに結果のみがすべてです．結果が合えばOK，あとは回路合成ツールがコンカレントな回路に変換してくれるに任せます．

えっ，回路合成ツールが合成不能を提示したらどうするかですって．それには，いくつもの対応策(？)があります．同じシーケンシャルな表現でも書き方を変えてみるとか．いろいろな回路合成ツールを評価して，自分の書いたコードを確実に合成してくれるツールを導入してみるとか，シーケンシャルな表現を諦めて，コンカレントな記述で階層設計をしてみるとか…(VHDLには多様な記述手段があって本当によかった)．

このように回路合成ツールがすべての記述に対応しているとは限らない現時点では，**設計者の柔軟な対応**が要求されるわけです．

設計者の柔軟な対応
設計環境がまだ完全とはいえない状況下においては，それを使う側の設計者が柔軟な対応をしなければならない．

●実回路化を意識するのはやめて，ビットごとの処理を同じ形に揃えること

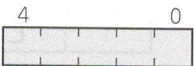
多くの場合，回路図は実回路化する場合を意識して最適化されて描かれています．このため，コンカレントな表現をする場合にはそのまま参照できもするでしょうが，シーケンシャルな表現をする場合は変形をして，ビットごとの処理が同じ形となるようにしたほうがよい場合があります．シーケンシャルな表現を行う際は，こうすれば伝搬遅延が減るといった，**回路設計上の良識**は一時棚上げしましょう．

回路設計上の良識
回路設計者が無意識のうちに行ってしまう，回路の伝搬遅延を少なくしたり，ゲート数を減らすような配慮のこと．

●変数は上書き可能であることを利用しよう

変数は，シーケンシャルな記述のなかで，幾度も書き換えをすることが可能です．このことを利用して，ビット間の情報伝達用の変数を使い回すことができます．たとえば，加算のビット間のキャリの受け渡しなども，ビット間ごとに変数を変えなくても一つの変数ですませることができます．

increment5ファンクションと5ビット・インクリメンタ

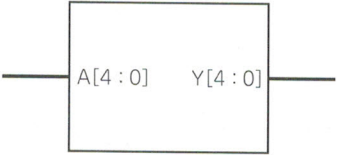
5ビット・バイナリ・データ上で＋1演算を行うincrement5関数を用いた，5ビット・インクリメンタについて解説します．

increment5の仕様を**表7-2**に，そして5ビット・インクリメンタの記号を**図7-11**に示します．

5ビット・インクリメンタは，5ビットの入力データに1を加えた値を出力とします．つまりincrement5ファンクションに**皮を被せただけの存在**であり，アーキテクチャ部の記述もincrement5ファンクションを用いた信号代入文1行です(**リスト7-4**)．

皮を被せただけの存在
このインクリメンタ回路の機能はincrement5関数の機能そのものである．VHDL記述自体も，increment5関数に機能モジュールの枠を被せただけのものとなっている．

〈表7-2〉increment5関数の仕様

書　式	increment5 (sorce)
引き数	4 … 0 (5ビット) : sorce
戻り値	4 … 0 (5ビット)
機　能	引き数の値に1を加えた結果を戻り値とする． 戻り値も5ビットなので，キャリが発生してもその情報は棄てられる．

〈図7-11〉5ビット・インクリメンタの記号

〈リスト7-4〉increment5ファンクションを用いた5ビット・インクリメンタ

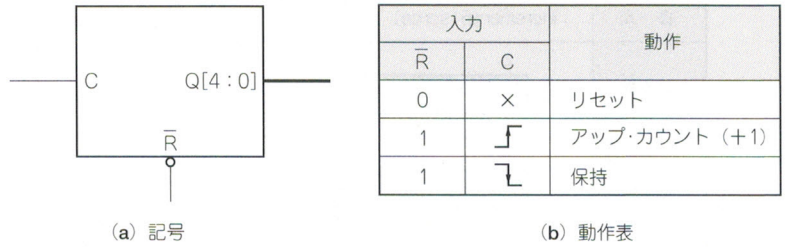
```
--
-- increment library
--
library ieee;
use ieee.std_logic_1164.all;
package libInc is
        function increment5(sorce  : std_logic_vector(4 downto 0)) return std_logic_vector;
end libInc;

package body libInc is
        function increment5(sorce  : std_logic_vector(4 downto 0)) return std_logic_vector is
                  variable andAll  : std_logic;
                  variable temp    : std_logic_vector(sorce'high downto sorce'low);
        begin
                  andAll := '1';
                  for j in sorce'low to sorce'high loop
                          temp(j)  := sorce(j) xor andAll;
                          andAll   := andAll and sorce(j);
                  end loop;
                  return(temp);
        end increment5;
end libInc;

--
-- 5-bit incrementor (use function)
--
library ieee;
use ieee.std_logic_1164.all;
use work.libInc.all;

entity inc5_func is
        port(     a : in std_logic_vector(4 downto 0);
                  y : out std_logic_vector(4 downto 0) );
end inc5_func;

architecture rtl of inc5_func is
begin

        y <= increment5(a);

end rtl;
```

注）increment関数のビット長を変更する場合は，上のリストのアミがかかっている部分を書き換える(5箇所)

〈図7-12〉5ビット・カウンタの仕様

（a）記号

入力		動作
R̄	C	
0	×	リセット
1	↑	アップ・カウント（+1）
1	↓	保持

（b）動作表

〈リスト7-5〉increment5ファンクションを用いた5ビット・カウンタ

```
--
-- increment library
--
library ieee;
use ieee.std_logic_1164.all;
package libInc is
        function increment5(sorce : std_logic_vector(4 downto 0)) return std_logic_vector;
end libInc;

package body libInc is
        function increment5(sorce : std_logic_vector(4 downto 0)) return std_logic_vector is
                  variable andAll : std_logic;
                  variable temp   : std_logic_vector(sorce'high downto sorce'low);
        begin
                  andAll := '1';
                  for j in sorce'low to sorce'high loop
                            temp(j) := sorce(j) xor andAll;
                            andAll := andAll and sorce(j);
                  end loop;
                  return(temp);
        end increment5;
end libInc;

--
-- 5-bit counter (use function)
--
library ieee;
use ieee.std_logic_1164.all;
use work.libInc.all;

entity count5_func is
        port(     c   : in std_logic;
                  nr  : in std_logic;
                  q   : out std_logic_vector(4 downto 0) );
end count5_func;

architecture rtl of count5_func is

        signal bufQ : std_logic_vector(4 downto 0);

begin
        process(nr,c)
        begin
                  if (nr = '0') then
                            bufQ <= (others => '0');
                  elsif (c'event and c = '1') then
                            bufQ <= increment5(bufQ);
                  end if;
        end process;

        q <= bufQ;

end rtl;
```

increment5ファンクションを用いた5ビット・カウンタ

5ビット(アップ)カウンタの仕様を**図7-12**に示します．

5ビット・アップ・カウンタの実現方法には，いくつかの手段がありますが，その一つが，Dレジスタとインクリメンタ(＋1加算器)を組み合わせるという方法です．今回は第5章で解説したDレジスタの記述とincrement5関数の組み合わせにより，5ビット・カウンタを実現しています．

機能モジュールの出力信号qは，方向性指定がoutであるため，qの値を機能モジュールの中から参照することはできません．このため，別途宣言した内部信号bufQを使って内部のフィードバック・ループを形成し，別途bufQの値を信号代入文で出力信号qへ代入しています(**リスト7-5**)．

5ビット(アップ)カウンタ
5ビットのバイナリ出力をもち，クロックが入るたびに出力が＋1されていく回路．

decrement8ファンクションと制御入力付き8ビット・デクリメンタ

8ビット・バイナリ・データ上で－1演算を行うdecrement8関数を用いた，制御入力付き8ビット・**デクリメンタ**について解説します．

decrement8の仕様を**表7-3**に，制御入力付き8ビット・デクリメンタの記号を**図7-13**に示します．

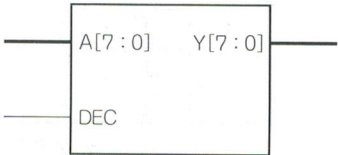
制御入力付き8ビット・デクリメンタは，DEC入力が1のときには入力データより1を引いた値を出力します．DEC入力が0のときには入力データをそのまま出力とします．

コード上ではwhen～else文を使って，入力データとdecrement8関数で処理した入力データを切り換えています(**リスト7-6**)．

デクリメンタ
－1演算を行う回路．

〈表7-3〉decrement8関数の仕様

書　式	decrement8(sorce)
引き数	7　…　0 (8ビット) : sorce
戻り値	7　…　0 (8ビット)
機　能	引き数の値より1を引いた結果を戻り値とする． 戻り値も8ビットなので，ボローが発生してもその情報は棄てられる．

〈図7-13〉制御入力付き8ビット・デクリメンタの記号

〈リスト7-6〉decrement8ファンクションを用いた制御入力付き8ビット・デクリメンタ

```
--
-- decrement library
--
library ieee;
use ieee.std_logic_1164.all;

package libDec is
        function decrement8(sorce : std_logic_vector(7 downto 0)) return std_logic_vector;
end libDec;

package body libDec is
        function decrement8(sorce : std_logic_vector(7 downto 0)) return std_logic_vector is
                variable orAll  : std_logic;
                variable temp   : std_logic_vector(sorce'high downto sorce'low);
        begin
                orAll := '0';
                for j in sorce'low to sorce'high loop
                        temp(j) := sorce(j) xor (not orAll);
                        orAll := orAll or sorce(j);
                end loop;
                return(temp);
        end decrement8;
end libDec;

--
-- 8-bit decrementor with control (use function)
--
library ieee;
use ieee.std_logic_1164.all;
use work.libDec.all;

entity dec8c_func is
        port(   a   : in std_logic_vector(7 downto 0);
                dec : in std_logic;
                y   : out std_logic_vector(7 downto 0) );
end dec8c_func;

architecture rtl of dec8c_func is
begin

        y <= decrement8(a) when (dec = '1') else a;

end rtl;
```

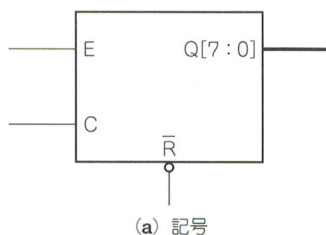
注）decrement関数のビット長を変更する場合は，上のリストのアミがかかっている部分を書き換える（5箇所）

decrement8ファンクションを用いたイネーブル付き8ビット・ダウン・カウンタ

イネーブル付き8ビット・ダウンカウンタの仕様を**図7-14**に示します．

カウンタの実現方法は，前出の5ビット・カウンタの場合と同じです．ただし，

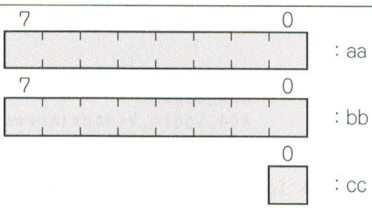
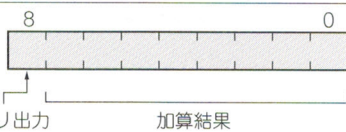
〈図7-14〉イネーブル付き8ビット・ダウン・カウンタの仕様

（a）記号

入力			動作
R̄	C	E	
0	×	×	リセット
1	↑	1	ダウン・カウント（−1）
1	↑	0	保持
1	↓	×	保持

（b）動作表

イネーブル機能を付加するために，if～then～else文による条件判断が追加されています(**リスト7-7**).

〈リスト7-7〉decrement8ファンクションを用いたイネーブル付き8ビット・ダウン・カウンタ

```
--
-- decrement library
--
library ieee;
use ieee.std_logic_1164.all;

package libDec is
        function decrement8(sorce  : std_logic_vector(7 downto 0)) return std_logic_vector;
end libDec;

package body libDec is
        function decrement8(sorce  : std_logic_vector(7 downto 0)) return std_logic_vector is
                variable orAll   : std_logic;
                variable temp    : std_logic_vector(sorce'high downto sorce'low);
        begin
                orAll := '0';
                for j in sorce'low to sorce'high loop
                        temp(j) := sorce(j) xor (not orAll);
                        orAll := orAll or sorce(j);
                end loop;
                return(temp);
        end decrement8;
end libDec;

--
-- 8-bit down counter with enable (use function)
--
library ieee;
use ieee.std_logic_1164.all;
use work.libDec.all;

entity count8de_func is
        port(   e  : in std_logic;
                c  : in std_logic;
                nr : in std_logic;
                q  : out std_logic_vector(7 downto 0) );
end count8de_func;

architecture rtl of count8de_func is

        signal bufQ : std_logic_vector(7 downto 0);

begin
        process(nr,c)
        begin
                if (nr = '0') then
                        bufQ <= (others => '0');
                elsif (c'event and c = '1') then
                        if (e = '1') then
                                bufQ <= decrement8(bufQ);
                        end if;
                end if;
        end process;

        q <= bufQ;

end rtl;
```

adc8ファンクションと8ビット加算器

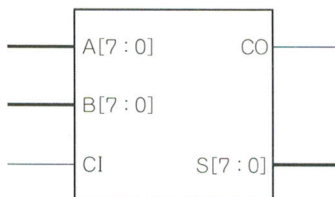
キャリ
桁上げ信号.

これまで解説した関数は引き数を一つしかもっていませんでしたが，ここで複数の引き数をもつ関数が登場します．adc8ファンクションは，8ビットの二つのデータの加算を行う関数であり，下位桁よりの桁上げに対応するための**キャリ**入力にも対応しています(**表7-4**)．adc8ファンクションには8ビット・データの加算にかかわるすべての機能があるので，これを利用して加算器を作ってみます．

adc8ファンクションを用いて作ろうとする加算器の記号を**図7-15**に示します．コードでは，A,B,CI入力の値をadc8関数に引き数として与え，得られた加算結果のビット8をCO出力にビット7～0をS出力にしています(**リスト7-8**)．

〈表7-4〉adc8関数の仕様

書　式	adc8(aa,bb,cc)
引き数	7 … 0 ：aa 7 … 0 ：bb 0 ：cc
戻り値	8 … 0 キャリ出力　加算結果
機　能	aa,bb(それぞれ8ビットのバイナリ・データ)およびcc(1ビット・データ)を足し上げた結果を9ビットのバイナリ・データとして返す. 戻り値はビット拡張された演算結果として見ることもできるし，8ビットの加算結果(ビット7～0)の上位にキャリ出力(ビット8)を合成したものとみなすこともできる.

〈図7-15〉8ビット加算器の記号

add4ファンクションと＋nカウンタ

インクリメント・ファンクション
＋1演算を行う関数のこと.

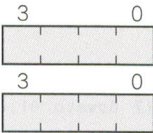
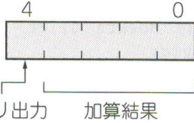
add4ファンクションは，4ビットの二つのデータの加算を行う関数です(**表7-5**)．adcファンクションと異なっているのは，下位よりの桁上げのためのキャリ入力をもたない点です．適用例としては，**インクリメント・ファンクション**と組み合わせて，＋nカウンタを実現してみました．

add4ファンクションを用いて実現しようとする＋nカウンタは**図7-16**のよう

〈リスト7-8〉adc8ファンクションを用いた8ビット加算器

```
--
-- add (with carry input) library
--
library ieee;
use ieee.std_logic_1164.all;
package libAdc is
        function adc8(aa,bb : std_logic_vector(7 downto 0);cc : std_logic) return std_logic_vector;
end libAdc;

package body libAdc is
        function adc8(aa,bb : std_logic_vector(7 downto 0);cc : std_logic) return std_logic_vector is
                variable interCarry : std_logic;
                variable temp       : std_logic_vector(aa'high + 1 downto aa'low);
        begin
                interCarry := cc;
                for j in aa'low to aa'high loop
                        temp(j)     := aa(j) xor bb(j) xor interCarry;
                        interCarry := (aa(j) and bb(j)) or ( (aa(j) or bb(j)) and interCarry);
                end loop;
                temp(aa'high + 1) := interCarry;
                return(temp);
        end adc8;
end libAdc;

--
-- adder8 (use function)
--
library ieee;
use ieee.std_logic_1164.all;
use work.libAdc.all;

entity adder8_func is
        port(   a  : in std_logic_vector(7 downto 0);
                b  : in std_logic_vector(7 downto 0);
                ci : in std_logic;
                co : out std_logic;
                s  : out std_logic_vector(7 downto 0) );
end adder8_func;

architecture rtl of adder8_func is

        signal temp : std_logic_vector(8 downto 0);

begin

        temp <= adc8(a,b,ci);

        co <= temp(8);
        s  <= temp(7 downto 0);

end rtl;
```

注）adc関数のビット長を変更する場合は，上のリストのアミがかかっている部分を書き換える（5箇所）

なものです．

ここで取り上げる＋nカウンタは，10ビットの出力をもつアップ・カウンタです．非同期リセット入力$\overline{R}$，クロックCおよび**同期イネーブル入力**Eのほかに，4ビット・データ入力Nをもっているのが特徴です．N以外の入力は，これまでに登場した同期イネーブル付きカウンタのそれとまったく同じ働きをします．異なっているのはカウントの進み方です．

$\overline{R}$入力およびE入力が1の場合，このカウンタはカウント動作を行いますが，その場合，Q出力はNに設定した値ずつ進んでいきます．たとえばNの設定が3の場合には，出力Qの値は3ずつ進んでいきます（クロックの立ち上がりごと

同期イネーブル入力
この場合は，クロックに同期してカウント動作を行うか行わないかを制御するための入力のこと．

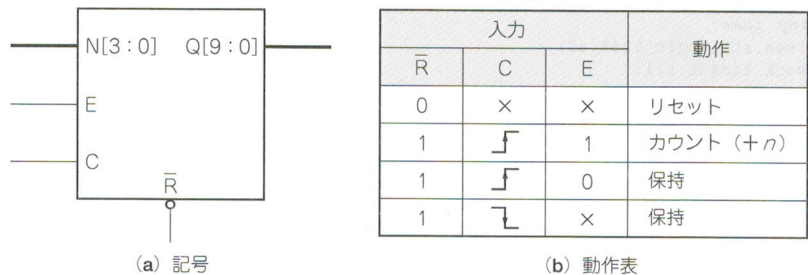
〈表7-5〉add4関数の仕様

項目	内容
書　式	add4(aa,bb)
引き数	3 … 0 : aa 3 … 0 : bb
戻り値	4 … 0 キャリ出力　加算結果
機　能	aaとbbの各4ビットのバイナリ・データ間で加算を行い，その結果を5ビットの戻り値として返す． 戻り値はビット拡張された加算結果，ないしは4ビットの加算結果の上位桁へのキャリ出力を付加したものとなる．

〈図7-16〉＋nカウンタの仕様

(a) 記号

入力			動作
$\overline{R}$	C	E	
0	×	×	リセット
1	↑	1	カウント（＋n）
1	↑	0	保持
1	↓	×	保持

(b) 動作表

に＋3される)．

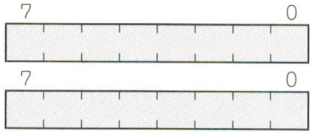
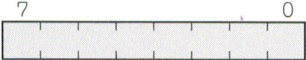
このような働きを実現するために，ここではカウント・データを上位6ビットと下位4ビットに分けて取り扱っています．add4関数を使ってN入力とQ出力の下位4ビットの加算を行い，その結果を次回のQ出力(下位4ビット)の値としています．上位6ビットの取り扱いは，add4関数の戻り値のビット4(上位桁へのキャリに相当)の値に左右されます．次回のQ出力(上位6ビット)の値は，add4の戻り値のビット4が1の時に，現在のQ出力(上位6ビット)に1を加えたものとなり，0のときには現在の出力と同じ値を保持します．実際のコードではこのあたりの処理をincrement6関数とwhen～else文により行っています(**リスト7-9**)．

〈リスト7-9〉add4ファンクションを用いた＋nカウンタ

```
--
-- increment library
--
library ieee;
use ieee.std_logic_1164.all;
package libInc is
        function increment6(sorce  : std_logic_vector(5 downto 0)) return std_logic_vector;
end libInc;

package body libInc is
        function increment6(sorce  : std_logic_vector(5 downto 0)) return std_logic_vector is
                 variable andAll  : std_logic;
                 variable temp    : std_logic_vector(sorce'high downto sorce'low);
        begin
                 andAll := '1';
                 for j in sorce'low to sorce'high loop
                         temp(j)  := sorce(j) xor andAll;
                         andAll  := andAll and sorce(j);
                 end loop;
                 return(temp);
        end increment6;
end libInc;

--
-- add library
--
library ieee;
use ieee.std_logic_1164.all;

package libAdd is
        function add4(aa,bb : std_logic_vector(3 downto 0)) return std_logic_vector;
end libAdd;

package body libAdd is
        function add4(aa,bb : std_logic_vector(3 downto 0)) return std_logic_vector is
                 variable interCarry  : std_logic;
                 variable temp        : std_logic_vector(aa'high + 1 downto aa'low);
        begin
                 interCarry := '0';
                 for j in aa'low to aa'high loop
                         temp(j)     := aa(j) xor bb(j) xor interCarry;
                         interCarry  := (aa(j) and bb(j)) or ( (aa(j) or bb(j)) and interCarry);
                 end loop;
                 temp(aa'high + 1)   := interCarry;
                 return(temp);
        end add4;
end libAdd;

--
-- +n counter with enable (use function)
--
-- data length : 10-bit  /  variable input : 4-bit
--
library ieee;
use ieee.std_logic_1164.all;
use work.libInc.all;
use work.libAdd.all;

entity count10_4var_func is
        port(     n  : in std_logic_vector(3 downto 0);
                  e  : in std_logic;
                  c  : in std_logic;
                  nr : in std_logic;
                  q  : out std_logic_vector(9 downto 0) );
end count10_4var_func;

architecture rtl of count10_4var_func is
```

注）add関数のビット長を変更する場合は，上のリストのアミがかかっている部分を書き換える(5箇所)

〈リスト7-9〉add4ファンクションを用いた十nカウンタ(つづき)

```
        signal sum          : std_logic_vector(4 downto 0);
        signal interCarry   : std_logic;
        signal bufQIn       : std_logic_vector(9 downto 0);
        signal bufQ         : std_logic_vector(9 downto 0);
        signal bufQ9downto4 : std_logic_vector(5 downto 0);

begin
        process(nr,c)
        begin
                if (nr = '0') then
                        bufQ <= (others => '0');
                elsif (c'event and c = '1') then
                        if (e = '1') then
                          bufQ <= bufQIn;
                        end if;
                end if;
        end process;

        sum                 <= add4(bufQ(3 downto 0),n);
        interCarry          <= sum(4);
        bufQ9downto4        <= bufQ(9 downto 4);
        bufQIn(9 downto 4) <= increment6(bufQ9downto4) when (interCarry = '1') else bufQ9downto4;
        bufQIn(3 downto 0) <= sum(3 downto 0);

        q <= bufQ;

end rtl;
```

fEq8/fGtb8ファンクションとコンパレータ

fEq8ファンクションは，8ビットの二つのデータが等しい場合に1を返す関数です(**表7-6**)．fGtb8ファンクションは二つの8ビット・データの大小比較結果を返す関数です(**表7-7**)．ここでは，これら二つの関数を用いて，二つのデータの比較を行うコンパレータを作ってみます．

これら二つのファンクションを用いて作ろうとするのは，**図7-17**のようなコンパレータです(**リスト7-10**)．

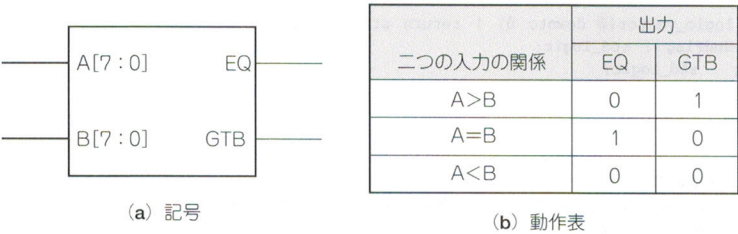
〈表7-6〉fEq8関数の仕様

書　式	fEq8(aa,bb)
引き数	7 … 0 [8ビット] : aa 7 … 0 [8ビット] : bb
戻り値	[1ビット]　1 : aa = bb　0 : aa ≠ bb
機　能	aaとbbの二つの8ビット・データを比較して，両者が一致した場合には1を，不一致の場合には0を戻り値として返す．

〈表7-7〉fGtb8関数の仕様

書　式	fGtb8(aa,bb)
引き数	7～0 : aa 7～0 : bb
戻り値	□　1 : aa > bb　　0 : aa ≦ bb
機　能	aaとbbの二つの8ビット・データを比較して，aaがbbよりも大きければ1を，それ以外の場合には0を戻り値として返す．

注）Gtbは（A）greater than Bの意味．

〈図7-17〉8ビット・コンパレータの仕様

（a）記号

二つの入力の関係	出力 EQ	出力 GTB
A>B	0	1
A=B	1	0
A<B	0	0

（b）動作表

〈リスト7-10〉fEq8/fGtb8ファンクションを用いたコンパレータ

```
--
-- compare library
--
library ieee;
use ieee.std_logic_1164.all;

package libComp is
	function fEq8(aa,bb  : std_logic_vector(7 downto 0) ) return std_logic;    -- return '1' when (aa = bb)
	function fGtb8(aa,bb  : std_logic_vector(7 downto 0) ) return std_logic;    -- return '1' when (aa > bb)
end libComp;

package body libComp is
	function fEq8(aa,bb : std_logic_vector(7 downto 0) ) return std_logic is
		variable notEqual : std_logic;
	begin
		notEqual := '0';
		for j in aa'high downto aa'low loop
			notEqual := notEqual or (aa(j) xor bb(j));
		end loop;
		return(not notEqual);
	end fEq8;

	function fGtb8(aa,bb : std_logic_vector(7 downto 0) ) return std_logic is
		variable compareEndFlag : std_logic;
		variable gtbFlag : std_logic;
	begin
		compareEndFlag := '0';
		gtbFlag := '0';
		for j in aa'high downto aa'low loop
			if (compareEndFlag = '0') then
				compareEndFlag := compareEndFlag or (aa(j) xor bb(j));
				gtbFlag := aa(j) and (not bb(j));
			end if;
		end loop;
		return(gtbFlag);
	end fGtb8;
end libComp;

--
-- 8-bit comparetor
--
library ieee;
use ieee.std_logic_1164.all;
use work.libComp.all;

entity comp8_func is
	port(	a   : in std_logic_vector(7 downto 0);
		b   : in std_logic_vector(7 downto 0);
		eq  : out std_logic;
		gtb : out std_logic);
end comp8_func;

architecture rtl of comp8_func is

begin

	eq  <= fEq8(a,b);
	gtb <= fGtb8(a,b);

end rtl;
```

注）fEqやfGtb関数のビット長を変更する場合は，上のリストのアミがかかっている部分を書き換える(9箇所)

第8章 VHDLの構文を使って機能回路を作る

各種の回路の記述

吉澤　清

それでは，これまで解説してきた構文を使って，いろいろな回路を記述してみましょう．

SIPOシフトレジスタ

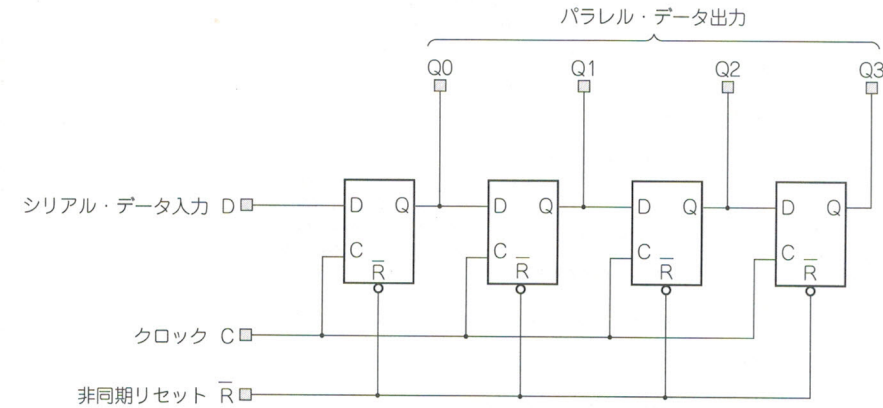
レジスタ回路の中でも，レジスタ内でのシリアルなデータの移動が可能なものを**シフトレジスタ**と呼びます．シフトレジスタも，ディジタル回路の基本回路の一つであり，この回路をベースにさまざまな回路が構成されます．

シフトレジスタの中でもいちばんスタンダードなものが，データを直列に取り込み，内部に取り込んだデータを並列に出力する，いわゆるSIPO(Serial-In Parallel-Out)シフトレジスタです(**図8-1**，**表8-1**)．

シフトレジスタ上のデータは，クロックが立ち上がるたびに1ビットずつ後段側へとシフトしていきます．同時に初段には，D入力の値が取り込まれます．

ここでコード例として示すのは，Dフリップフロップのコーポネントを用いた階層設計によるものです(**リスト8-1**)．

シフトレジスタ
データのパラレル-シリアル変換などに使われるほか，ジョンソン・カウンタ/LFSR/ワンホット・シーケンサなどの基本回路となっている．

〈図8-1〉基本的な4ビット・シフトレジスタ(SIPO)

〈表8-1〉4ビットSIPOシフトレジスタの真理値表

入力			出力				動作
$\overline{R}$	C	D	Q0	Q1	Q2	Q3	
0	×	×	0	0	0	0	リセット
1	↑	1	1	Q0*n*	Q1*n*	Q2*n*	シフト
1	↑	0	0	Q0*n*	Q1*n*	Q2*n*	
1	↓	×	Q0*n*	Q1*n*	Q2*n*	Q3*n*	ホールド

×：不定

〈リスト8-1〉4ビットSIPOシフトレジスタ

```
--
-- D-F.F. with asynchronous reset
--
library ieee;
use ieee.std_logic_1164.all;

entity df is
          port(    d   : in std_logic;
                   c   : in std_logic;
                   nr  : in std_logic;
                   q   : out std_logic);
end df;

architecture rtl of df is
begin

          process(nr,c)
          begin

                    if (nr = '0') then
                              q <= '0';
                    elsif (c'event and c = '1') then
                              q <= d;
                    end if;

          end process;

end rtl;

--
-- 4-bit shift-reg.
--
library ieee;
use ieee.std_logic_1164.all;

entity sr4 is
          port(    d   : in std_logic;
                   c   : in std_logic;
                   nr  : in std_logic;
                   q   : out std_logic_vector(3 downto 0) );
end sr4;

architecture rtl of sr4 is

          component df
                    port(    d   : in std_logic;
                             c   : in std_logic;
                             nr  : in std_logic;
                             q   : out std_logic);
          end component;

          signal bufQ : std_logic_vector(3 downto 0);

begin

          mod0 : df port map(
                    d   => d,
                    c   => c,
                    nr  => nr,
                    q   => bufQ(0) );

          mod1 : df port map(
                    d   => bufQ(0),
                    c   => c,
                    nr  => nr,
                    q   => bufQ(1) );

          mod2 : df port map(
                    d   => bufQ(1),
                    c   => c,
```

〈リスト8-1〉4ビットSIPOシフトレジスタ(つづき)

```
                    nr  => nr,
                    q   => bufQ(2) );

          mod3  : df port map(
                    d   => bufQ(2),
                    c   => c,
                    nr  => nr,
                    q   => bufQ(3) );

          q <= bufQ;

end rtl;
```

イネーブル付きSIPOシフトレジスタ

同期設計用に，前出のSIPOシフトレジスタに**イネーブル**(シフト・イネーブル)入力を追加してみましょう(**図8-2**，**表8-2**)．

イネーブル入力を設けることにより，クロックは入れっ放しでも，シフトのコントロールができるようになります．

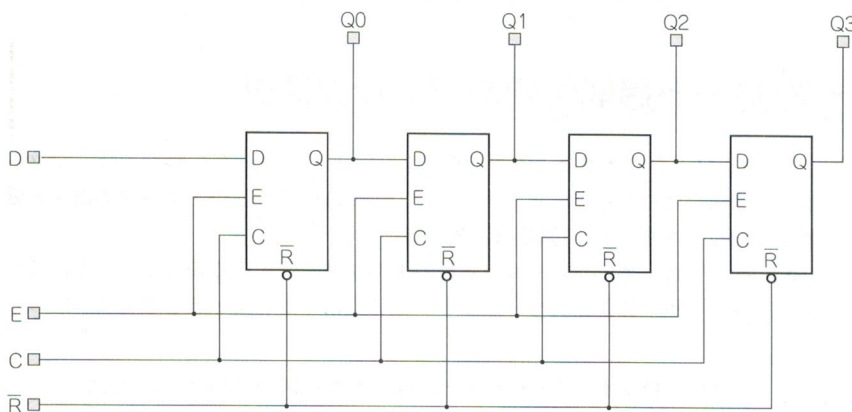
階層設計によれば，プリミティブなフリップフロップを使って，回路図通りのコードを書くこともできます．しかしここでは，**連接演算子**を用いたデータのシフトの表現とレシスタ記述の組み合わせによりコードを記述してみましょう．このほうがシフトレジスタのビット長が長くなった際に，短かいコードで済ませることが可能です(**リスト8-2**)．

イネーブル

前項のシフトレジスタでは，クロックの立ち上がりによりシフト動作が行われますが，シフトの制御はクロックの変化だけによっているため，同期設計用とはいえません．そこで，同期イネーブル入力を設け，この信号のレベルによりシフトの制御を行うようにした．

連接演算子

VHDLの演算子の一つ．"&"で表される．複数のデータを繋ぎ合わせる機能をもつ(第4章参照)．

〈図8-2〉4ビット・イネーブル付きSIPOシフトレジスタ

〈表8-2〉4ビット・イネーブル付きSIPOシフトレジスタの真理値表

入力				出力				動作
$\overline{R}$	C	E	D	Q0	Q1	Q2	Q3	
0	×	×	×	0	0	0	0	リセット
1	↑	1	1	1	Q0n	Q1n	Q2n	シフト
1	↑	1	0	0	Q0n	Q1n	Q2n	
1	↑	0	×	Q0n	Q1n	Q2n	Q3n	ホールド
1	↓	×	×	Q0n	Q1n	Q2n	Q3n	

X：不定

〈リスト8-2〉4ビット同期イネーブル付きSIPOシフトレジスタ

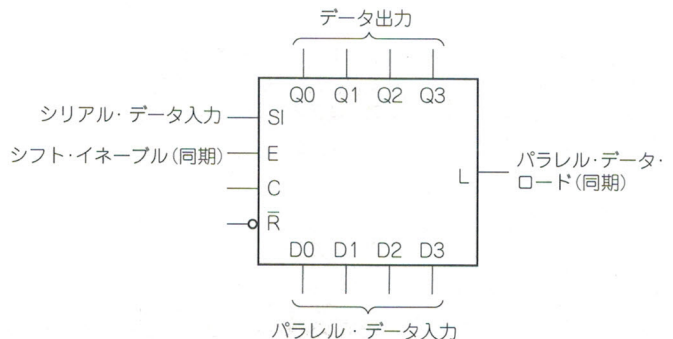
```
--
-- 4-bit shift-reg. with enable
--
library ieee;
use ieee.std_logic_1164.all;

entity sr4e is
        port(    d   : in std_logic;
                 e   : in std_logic;
                 c   : in std_logic;
                 nr  : in std_logic;
                 q   : out std_logic_vector(3 downto 0) );
end sr4e;

architecture rtl of sr4e is

        signal bufQ : std_logic_vector(3 downto 0);

begin

        process(nr,c)
        begin

                if (nr = '0') then
                        bufQ <= (others => '0');
                elsif (c'event and c = '1') then
                        if (e = '1') then
                                bufQ <= bufQ(2 downto 0) & d;
                        end if;
                end if;

        end process;

        q <= bufQ;

end rtl;
```

パラレル・データ・ロード機能付きシフトレジスタ

パラレル・データのロード機能
シフトレジスタへのデータの同期ロード機能のこと．ロード入力が‘1’のときに，クロックの立ち上がりでパラレル・データがレジスタに取り込まれる．

SIPOシフトレジスタは，シフト時に1ビットずつデータを取り込む，シフトインの機能だけをもっていますが，ここではこれに**パラレル・データのロード機能**を付加してみましょう(**図8-3，表8-3**)．

if～then～else文においては，複数の条件が書かれている場合には，先に書かれた条件より順に判断されていきます．ここではパラレル・データ・ロードのほ

〈図8-3〉4ビット・パラレル・データ・ロード機能付きシフトレジスタの記号

〈表8-3〉4ビット・パラレル・データ・ロード機能付きシフトレジスタの真理値表

入力					出力				動作
$\bar{R}$	C	L	E	SI	Q0	Q1	Q2	Q3	
0	×	×	×	×	0	0	0	0	リセット
1	↑	1	×	×	D0	D1	D2	D3	パラレル·データ·ロード(同期)
1	↑	0	1	1	1	Q0*n*	Q1*n*	Q2*n*	シフト
1	↑	0	1	0	0	Q0*n*	Q1*n*	Q2*n*	
1	↑	0	0	×	Q0*n*	Q1*n*	Q2*n*	Q3*n*	ホールド
1	↓	×	×	×	Q0*n*	Q1*n*	Q2*n*	Q3*n*	

×：不定

うが，データのシフトより優先順位が高いため，E入力に関する条件よりもL入力に関する条件のほうを先に記述しています(**リスト8-3**).

〈リスト8-3〉4ビット・パラレル・データ・ロード機能付きシフトレジスタ

```
--
-- 4-bit shift-reg. with synchronous parallel load & enable
--
library ieee;
use ieee.std_logic_1164.all;

entity sr4le is
        port(   d   : in std_logic_vector(3 downto 0);     -- parallel load data
                l   : in std_logic;                        -- paralell load
                si  : in std_logic;                        -- serial input data
                e   : in std_logic;                        -- shift enable
                c   : in std_logic;
                nr  : in std_logic;
                q   : out std_logic_vector(3 downto 0) );
end sr4le;

architecture rtl of sr4le is

        signal bufQ : std_logic_vector(3 downto 0);

begin

        process(nr,c)
        begin

                if (nr = '0') then
                        bufQ <= (others => '0');
                elsif (c'event and c = '1') then
                        if (l = '1') then
                                bufQ <= d;
                        elsif (e = '1') then
                                bufQ <= bufQ(2 downto 0) & si;
                        end if;
                end if;

        end process;

        q <= bufQ;

end rtl;
```

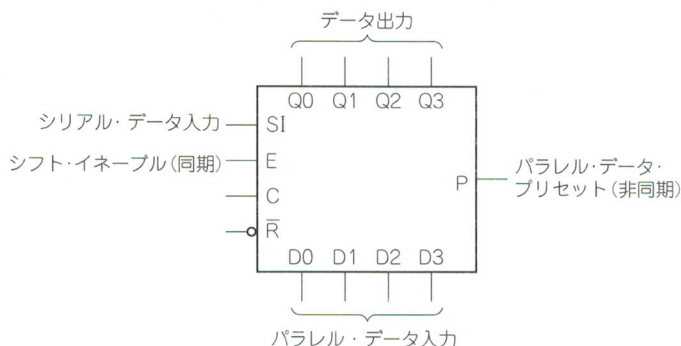
非同期パラレル・データ・プリセット機能付きシフトレジスタ

非同期でプリセット
パラレル・データをクロックの変化とは無関係にレジスタに取り込むこと．

前項では，同期パラレル・データ・ロードを行うシフトレジスタについて解説しましたが，次はパラレル・データを**非同期でプリセット**(ロード)するタイプのシフトレジスタです(**図8-4**，**表8-4**)．

カウンタへのデータ・ロード(同期/非同期)に関しても，同様の書き方が使えるので参考としてください．

コードを書く際に注意しなければならないのは，プリセット入力が非同期である点です．プリセット入力に関連するP入力とD入力は，プロセス文のセンシティビティ・リストに含める必要があります(**リスト8-4**)．

〈図8-4〉4ビット非同期パラレル・データ・プリセット機能付きシフトレジスタの記号

〈表8-4〉4ビット非同期パラレル・データ・プリセット機能付きシフトレジスタの真理値表

入力					出力				動作
R̄	P	C	E	SI	Q0	Q1	Q2	Q3	
0	×	×	×	×	0	0	0	0	リセット
1	1	×	×	×	D0	D1	D2	D3	パラレル・データ・プリセット(非同期)
1	0	↑	1	1	1	Q0n	Q1n	Q2n	シフト
1	0	↑	1	0	0	Q0n	Q1n	Q2n	
1	0	↑	0	×	Q0n	Q1n	Q2n	Q3n	ホールド
1	0	↓	×	×	Q0n	Q1n	Q2n	Q3n	

×：不定

〈リスト8-4〉4ビット非同期パラレル・データ・プリセット機能付きシフトレジスタ

```
--
-- 4-bit shift-reg. with asynchronous parallel preset & enable
--
library ieee;
use ieee.std_logic_1164.all;

entity sr4pe is
        port(   d   : in std_logic_vector(3 downto 0);   -- parallel preset data
                p   : in std_logic;                       -- asynchronous paralell preset
                si  : in std_logic;                       -- serial input data
                e   : in std_logic;                       -- shift enable
                c   : in std_logic;
                nr  : in std_logic;
                q   : out std_logic_vector(3 downto 0) );
end sr4pe;

architecture rtl of sr4pe is

        signal bufQ : std_logic_vector(3 downto 0);

begin

        process(nr,p,d,c)
        begin

                if (nr = '0') then
                        bufQ <= (others => '0');
                elsif (p = '1') then
                        bufQ <= d;
                elsif (c'event and c = '1') then
                        if (e = '1') then
                                bufQ <= bufQ(2 downto 0) & si;
                        end if;
                end if;

        end process;

        q <= bufQ;

end rtl;
```

双方向シフトレジスタ

CPUのアキュムレータなどにおいては片方向だけでなく，データを双方向にシフトすること(右シフト/左シフト)が必要になります．このような機能を実現してしくれるのが双方向シフトレジスタです．双方向シフトレジスタを実現するためには，各段のDフリップフロップの入力を**マルチプレクサ**により，プリセット用のデータ入力や前段のDフリップフロップの出力や後段のDフリップフロップの出力，そして自段の出力などに切り替えることができればよいわけです．

これにより，パラレル・データのロードや左シフト，右シフトやホールド(データ保持)などを実現することができます(**図8-5，表8-5，図8-6，リスト8-5**)．

これまでのシフトレジスタの解説では，図の左側にデータのLSB，そして図の右側にデータのMSBを配してきました．これはフリップフロップを組み合わせて回路図を描く場合には，データが左側より右側へ流れるように描くのが自然であることと，シリアル・データ入力が始めに取り込まれるのがビット0であるほうがより自然であることによっていました．

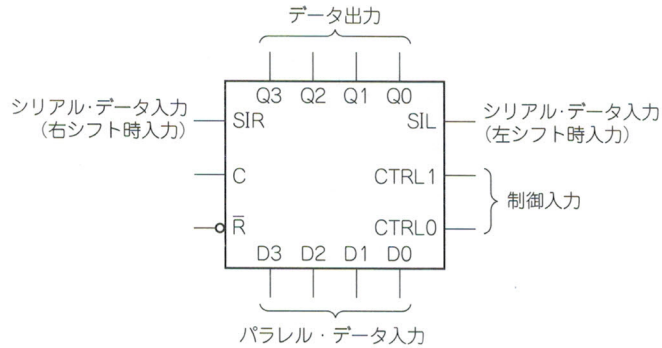
しかし，一般的に2進数の数値を表現する場合には，左側に**MSB**を，そして

マルチプレクサ
(データの)多重化器．データ・セレクタと同じ(第4章参照)．

MSB
データの最上位ビット．

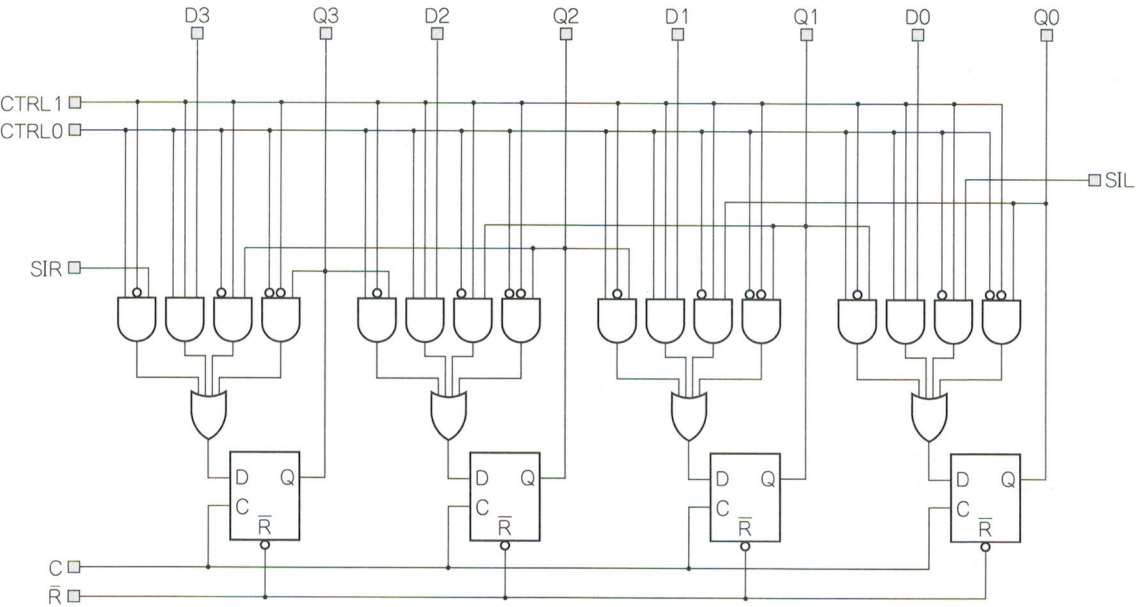
LSB
データの最下位ビット．

右側に**LSB**を配置するので，本項ではデータ配置をこれに合わせました．これは，一般的なデータの右シフトおよび左シフトの定義が，MSBが左，LSBが右

〈図8-5〉4ビット双方向シフトレジスタの記号

〈表8-5〉4ビット双方向シフトレジスタの真理値表

入力						出力				動作
$\overline{R}$	C	CTRL1	CTRL0	SIR	SIL	Q3	Q2	Q1	Q0	
0	×	×	×	×	×	0	0	0	0	リセット
1	↑	1	1	×	×	D3	D2	D1	D0	パラレル·データ·ロード
1	↑	1	0	×	1	Q2n	Q1n	Q0n	1	左シフト
1	↑	1	0	×	0	Q2n	Q1n	Q0n	0	
1	↑	0	1	1	×	1	Q3n	Q2n	Q1n	右シフト
1	↑	0	1	0	×	0	Q3n	Q2n	Q1n	
1	↑	0	0	×	×	Q3n	Q2n	Q1n	Q0n	ホールド
1	↓	×	×	×	×	Q3n	Q2n	Q1n	Q0n	

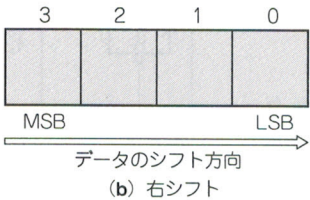
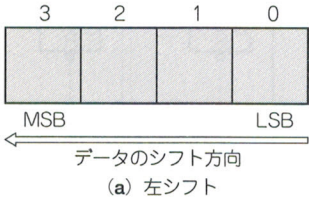
×：不定

〈図8-6〉4ビット双方向シフトレジスタの回路

〈リスト8-5〉4ビット双方向シフトレジスタ

```
--
-- synchronous bidirectional 4-bit shift-reg.
--

-- *** control signal table ***********
--
-- ctrl1    ctrl0     function
--    1         1        load parallel data
--    1         0        shift left
--    0         1        shift right
--    0         0        hold
--
-- ***********************************

library ieee;
use ieee.std_logic_1164.all;

entity sr4_bidir is
          port(    d     : in std_logic_vector(3 downto 0);   -- parallel load data
                   sir   : in std_logic;                      -- serial input for right-shift
                   sil   : in std_logic;                      -- serial input for left-shift
                   ctrl  : in std_logic_vector(1 downto 0);   -- control
                   c     : in std_logic;
                   nr    : in std_logic;
                   q     : out std_logic_vector(3 downto 0) );
end sr4_bidir;

architecture rtl of sr4_bidir is

          signal bufQ : std_logic_vector(3 downto 0);

begin

          process(nr,c)
          begin

                    if (nr = '0') then
                              bufQ <= (others => '0');
                    elsif (c'event and c = '1') then
                              if (ctrl = "11") then
                                        bufQ <= d;
                              elsif (ctrl = "10") then
                                        bufQ <= bufQ(2 downto 0) & sil;
                              elsif (ctrl = "01") then
                                        bufQ <= sir & bufQ(3 downto 1);
                              end if;
                    end if;

          end process;

          q <= bufQ;

end rtl;
```

〈図8-7〉データのシフト方向の定義

(a) 左シフト

(b) 右シフト

というデータ配置に基づいているためです(**図8-7**，解説の全般でデータの方向性が統一できなかったことについてはお許しください)．

したがって，これまでに解説してきたシフトレジスタは，この定義によれば左シフトをしていたことになります．

ジョンソン・カウンタ

シフトレジスタをベース
基本回路として，シフトレジスタを使ったという意味．

内部状態の数
カウンタなどにおいて，回路が取り得る出力パターンの数を指す．

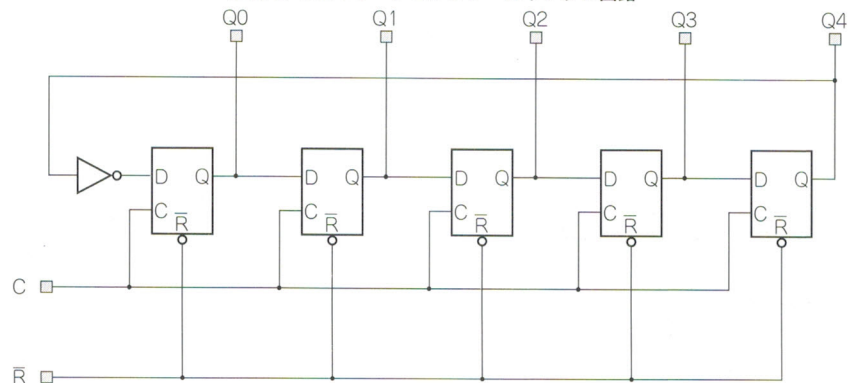
非同期リセット
クロック信号の変化とは無関係に，リセット入力のレベルによりレジスタの値をall'0'にする機能．

同期カウンタの多くはレジスタをベースとしますが，ジョンソン・カウンタは珍しく**シフトレジスタをベース**としています．ジョンソン・カウンタは，フィードバック回路が単純でほかのカウンタにくらべて高速動作が可能です．また，出力コードはデータの途中に1箇所変化点をもつだけなので，出力のデコードを行う回路も簡単なもので済ますことができます．

このことからもジョンソン・カウンタは高速で動作させようとする回路に向いていると言えます．さらに，先にも触れたように，出力コードの途中のデータの変化点は1箇所だけということで，動作を行う際に同時に反転するフリップフロップは1個だけということになります．このため，電源ラインへの放射ノイズも少なくなるものと考えられます．と，ここまでよいことばかりを述べてきましたが，よいことづくめに思われるジョンソン・カウンタにも欠点はあります．nステージ(フリップフロップの個数)のジョンソン・カウンタがもてる**内部状態**(出力パターン)**の数**は，その2倍($2n$状態)にしかなりません．nステージのバイナリ・カウンタの内部状態数は2^nとなることから，内部状態の数を多くしようとするとジョンソン・カウンタは回路量(フリップフロップの数)が増え不利となります．もっともこの点についても，複数のジョンソン・カウンタをシリーズ構成にするなどして改善を計ることが可能ではあります．

ジョンソン・カウンタは，シフトレジスタの最終段のデータを反転して，シフトレジスタのシリアル・データ入力にフィードバックをかけるという単純な構成をもちます(**図8-8**)．

なお，ジョンソン・カウンタの初期設定値には制限が付きます．ジョンソン・カウンタは，動作開始時に内部状況がall'0'またはall'1'あるいはデータの途中で1回だけ値が変化するデータ(たとえば"11000"とか"00001"とか)でなければなりません．きちんと初期設定がなされなかった場合，ジョンソン・カウンタは正常な動作をすることができなくなります．**図8-8**の回路では全段に**非同期リセット**がかけられるので，これにより初期状態を"00000"にしたのち，カウンタとし

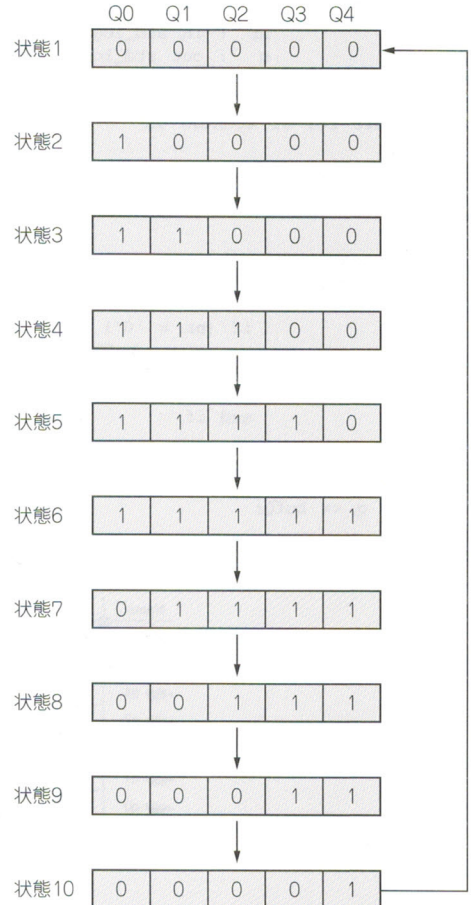
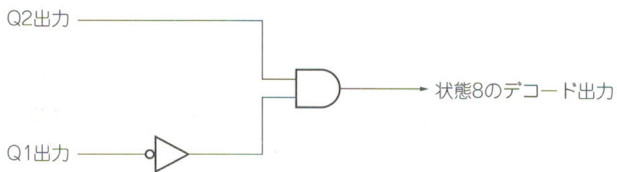
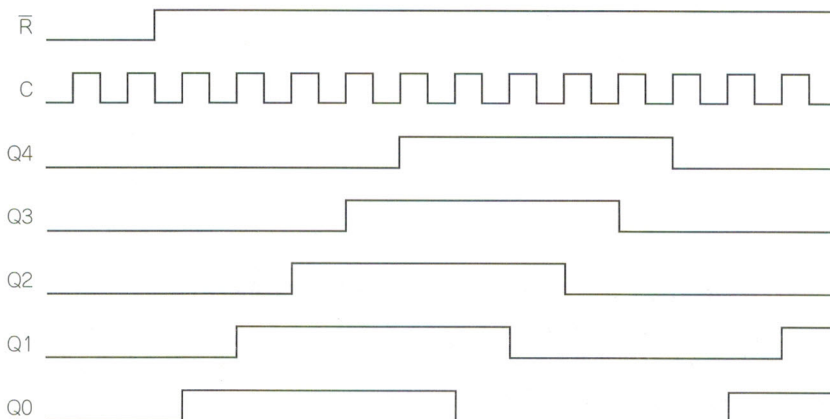
〈図8-8〉5ビット・ジョンソン・カウンタの回路

ての動作をさせます(**表8-6，図8-9，図8-10**).

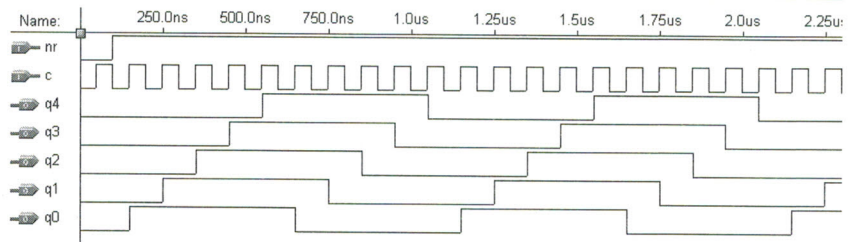
ジョンソン・カウンタの状態デコードは，2入力ANDゲートとインバータにより行うことができます．たとえば状態8の場合，Q4～Q0のデータは“11100”となります．Q2が1でQ1が0となる場合は，状態8以外にはあり得ないため，

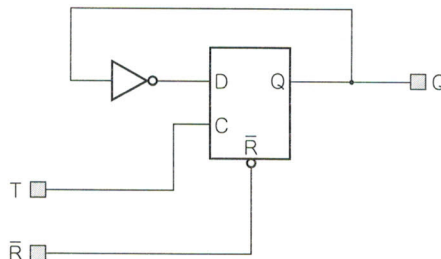
〈表8-6〉5ビット・ジョンソン・カウンタの真理値表

入力		出力					動作
$\overline{R}$	C	Q4	Q3	Q2	Q1	Q0	
0	×	0	0	0	0	0	リセット
1	↑	Q3*n*	Q2*n*	Q1*n*	Q0*n*	$\overline{Q4n}$	カウント
1	↓	Q4*n*	Q3*n*	Q2*n*	Q1*n*	Q0*n*	ホールド

×：不定

〈図8-9〉5ビット・ジョンソン・カウンタの内部状態の遷移

〈図8-11〉状態8のデコード回路

〈図8-10〉5ビット・ジョンソン・カウンタの動作タイミング・チャート

〈リスト8-6〉5ビット・ジョンソン・カウンタ

```
--
-- 5-bit jonson counter
--
library ieee;
use ieee.std_logic_1164.all;

entity countJ5 is
          port(    c   : in std_logic;
                   nr  : in std_logic;
                   q   : out std_logic_vector(4 downto 0));
end countJ5;

architecture rtl of countJ5 is

          signal bufQ : std_logic_vector(4 downto 0);

begin

          process(nr,c)
          begin

                    if (nr = '0') then
                              bufQ <= (others => '0');
                    elsif (c'event and c = '1') then
                              bufQ <= bufQ(3 downto 0) & (not bufQ(4));
                    end if;

          end process;

          q <= bufQ;

end rtl;
```

〈図8-12〉T型フリップフロップ

(a) 回路

入力		出力	動作
R̄	T	Q	
0	×	0	リセット
1	↑	$\overline{Q_n}$	反転
1	↓	Q_n	ホールド

×：不定

(b) 真理値表

この2本の出力をデコードすればよいことになります(**図8-11**).

ジョンソン・カウンタはシフトレジスタをベースとしているため，VHDLコードもシフトレジスタのそれとほぼ同じです．ここでは連接演算子を使った場合のコードを示します(**リスト8-6**).

考えてみると，D型フリップフロップの出力を反転して自らのD入力にフィー

ドバックするT型フリップフロップは，1ビットのジョンソン・カウンタともいうことができるのかもしれません(図8-12).

状態数も2であり，ジョンソン・カウンタの状態数の定義とも合致します.

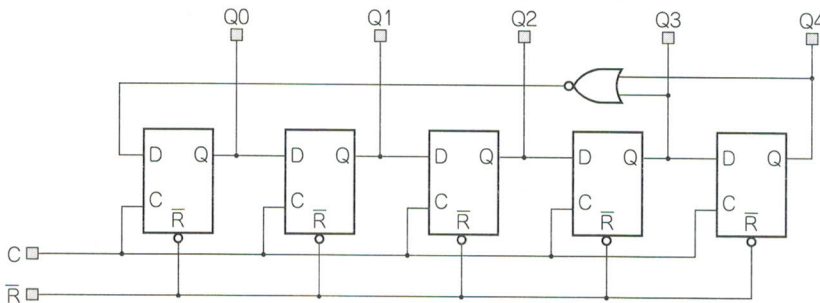
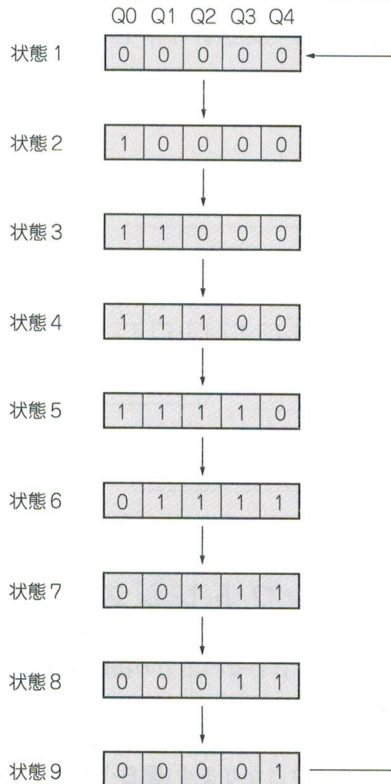
状態数が奇数となるジョンソン・カウンタ

前項において，ジョンソン・カウンタの状態数はステージ数の2倍になると説明しました．そうすると，必要となる状態数が4，6，8，…と偶数の場合にはよいのですが，**状態数を奇数としたい場合**にはどうすればよいのでしょうか.

ジョンソン・カウンタはシフトレジスタにフィードバックをかけたものです．このフィードバック回路をいじることにより細工を施すことはできないものでしょうか．通常のジョンソン・カウンタは，シフトレジスタの終段の出力を反転し

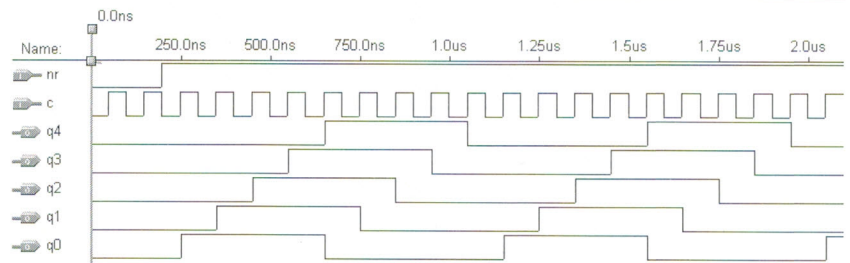
状態数を奇数としたい場合

ジョンソン・カウンタの状態数は基本的に2の倍数(偶数)となる．それでは，状態数を奇数としたい場合にはどうするかというのが，本項のテーマ.

〈図8-13〉5ビット9状態ジョンソン・カウンタの回路

〈図8-14〉5ビット9状態ジョンソン・カウンタの内部状態の遷移

て初段のデータ入力へとフィードバックしますが，終段の出力と終段から二つ目の出力のNORを取って初段のデータ入力へフィードバックをかけると，通常より状態数が一つ減って状態数を奇数にすることができます(**図8-13**).

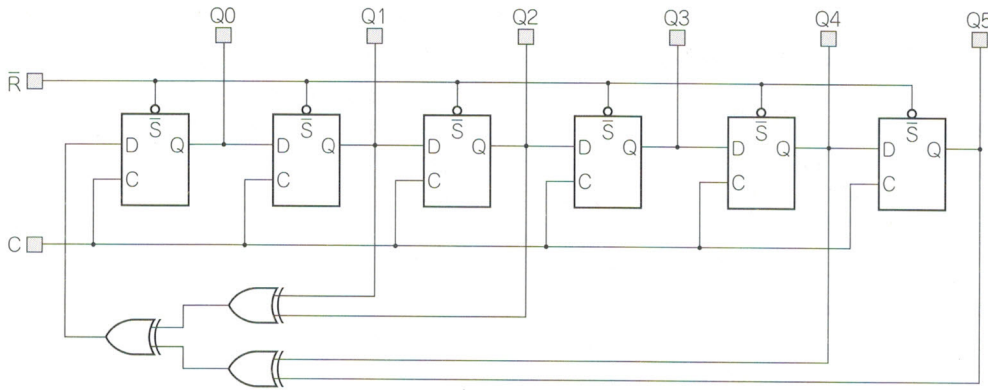
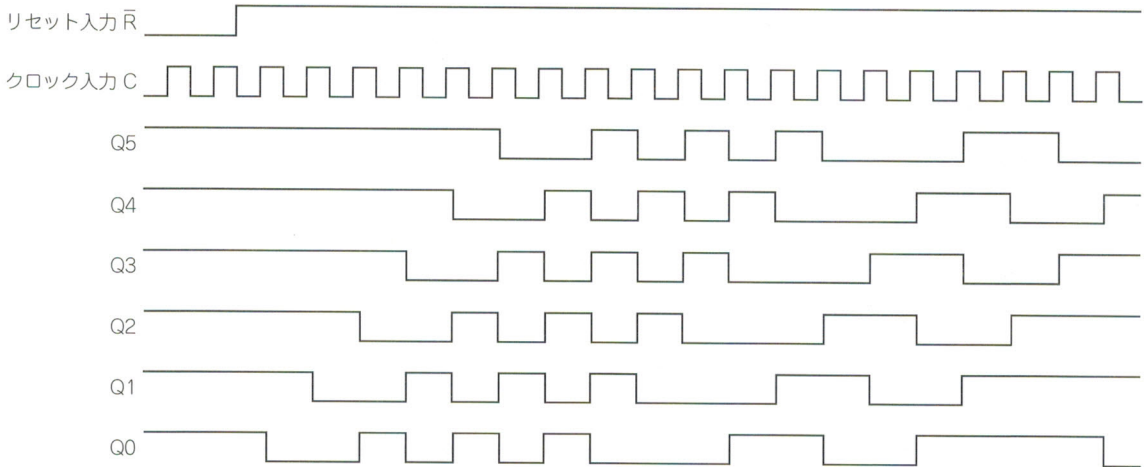
このような回路にすると，終段が'1'になる前，終段より2段目が'1'となった時点で初段のデータ入力の値が'0'となります．このため内部状態が"11111"(全段が'1')になることがなくなり("11111"の状態が飛ばされる)，状態数が一つ減る結果となります(**図8-14，リスト8-7**).

〈リスト8-7〉5ビット9状態ジョンソン・カウンタ

```
--
-- 5-bit jonson counter (9 state)
--
library ieee;
use ieee.std_logic_1164.all;

entity countJ5st9 is
        port(   c   : in std_logic;
                nr  : in std_logic;
                q   : out std_logic_vector(4 downto 0));
end countJ5st9;

architecture rtl of countJ5st9 is

        signal bufQ : std_logic_vector(4 downto 0);

begin

        process(nr,c)
        begin

                if (nr = '0') then
                        bufQ <= (others => '0');
                elsif (c'event and c = '1') then
                        bufQ <= bufQ(3 downto 0) & (bufQ(4) nor bufQ(3));
                end if;

        end process;

        q <= bufQ;

end rtl;
```

LFSR(リニア・フィードバック・シフトレジスタ)

排他的論理和
論理演算の一つ．Ex-OR回路により実現される(第4章参照).

シフトレジスタの特定の組み合わせの出力の**排他的論理和**を，シフトレジスタの入力に戻すことにより2^n-1(ただし，nはシフトレジスタのビット長)周期でパターンが一巡するカウンタが得られます．このような回路は**LFSR**と呼ばれ，その出力は**M系列**(乱数の一種)となります(**図8-15**).

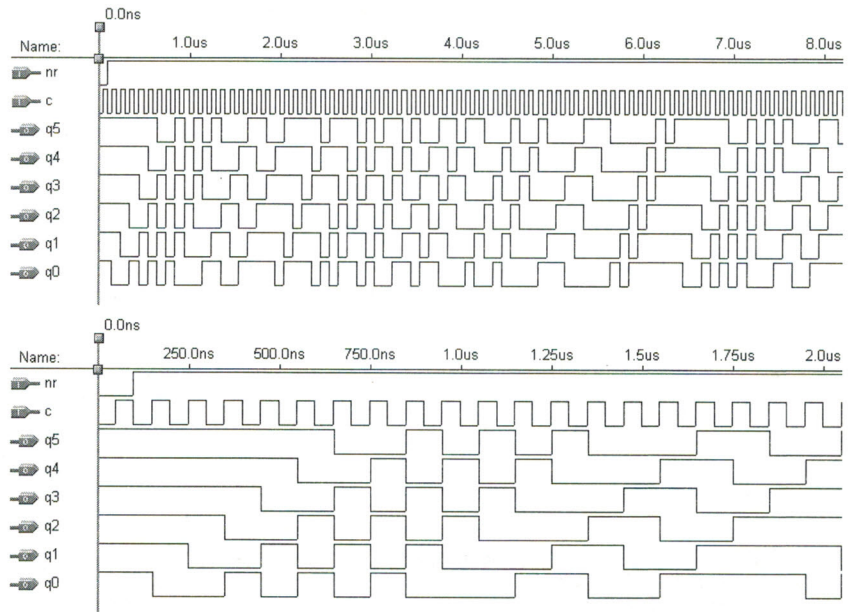
〈図8-15〉6ビットLFSR（帰還タップ＝(5, 4, 2, 1)）

〈図8-16〉6ビットLFSR（帰還タップ＝(5, 4, 2, 1)）の動作タイミング・チャート（部分）

この回路を運用する上で注意しなければならないのは，このカウンタの値がall0となると，カウンタはデッドロックに陥り動かなくなる点です．フィードバック用のデータがすべて0になると，それらの排他的論理和の出力も0となり，本質的にシフトレジスタであるLFSRはall0の状態を保持してしまうことになるわけです．LFSRは2^n-1の状態を繰り返しますが，除外されている1状態が，このすべての出力が0の状態です．

通常，カウンタ回路の初期設定はall0とされたりすることが多いのですが，LFSRの場合にはこれは厳禁です．ここでは非同期リセット入力により，全段をセットしており，all1の状態からカウントをスタートさせています（**図8-16**，**リスト8-8**）．

LFSR
Linear Feedback Shift Register．シフトレジスタにEx－ORを使って帰還をかけた回路．

M系列
Maximum length sequence．最大長系列．疑似不規則信号の一種．

〈リスト8-8〉6ビットLFSR

```
--
-- 6-bit LFSR(Linear Feedback Shift Register)
--
library ieee;
use ieee.std_logic_1164.all;

entity lfsr6a is
        port(   c   : in std_logic;
                nr  : in std_logic;
                q   : out std_logic_vector(5 downto 0));
end lfsr6a;

architecture rtl of lfsr6a is

        signal feedback : std_logic;
        signal bufQ     : std_logic_vector(5 downto 0);

begin

        process(nr,c)
        begin

                if (nr = '0') then
                        bufQ <= (others => '1');
                elsif (c'event and c = '1') then
                        bufQ <= bufQ(4 downto 0) & feedback;
                end if;

        end process;

-- feedback tap = (5,4,2,1)
        feedback <= bufQ(5) xor bufQ(4) xor bufQ(2) xor bufQ(1);

        q <= bufQ;

end rtl;
```

グレイ・コード・カウンタ

グレイ・コードは，隣接するコード間でコードの値が1ビットしか変わらないようなコード体系のことを言います．

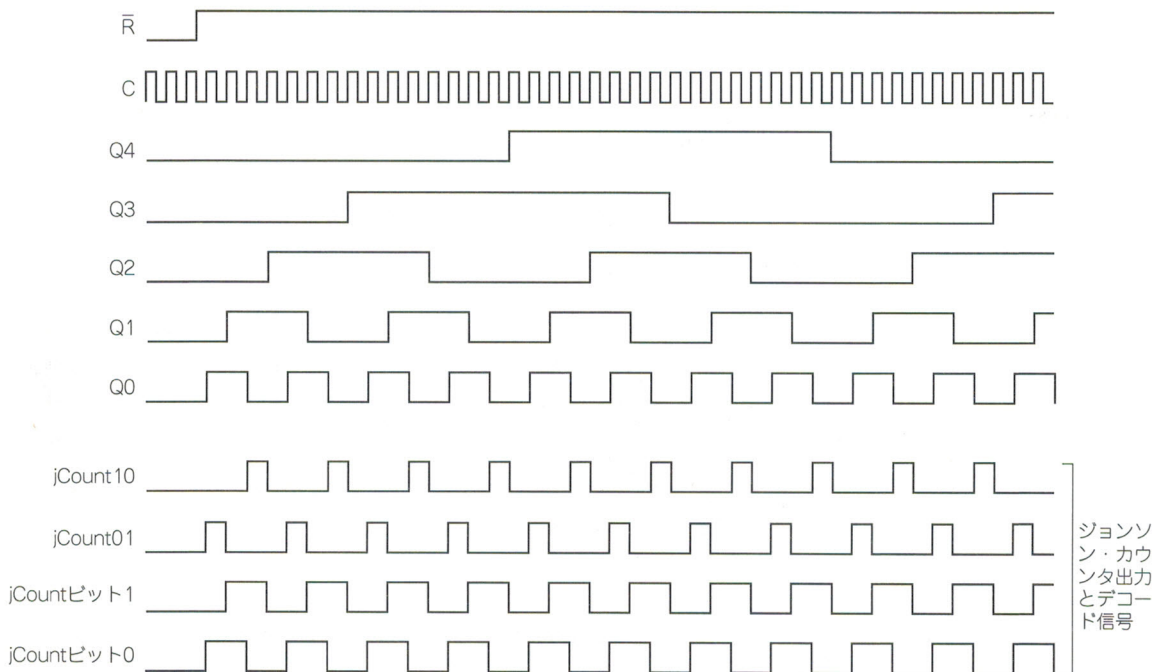
グレイ・コードによりカウントを行うカウンタのことをグレイ・カウンタと呼び，動作時に，反転するビットが一時に1ビットだけであることから，電源ラインへ放射するノイズが少なくなり，また，出力のデコードを行う場合に**グリッチ**が発生しないなどのメリットをもっています．

グレイ・コード・カウンタの設計の仕方も，いろいろあるようですが，ここではグレイ・コード・カウンタの出力パターンの規則性より，グレイ・コードでカウントを行うカウンタを作ってみたいと思います．

グレイ・コード・カウンタのLSB(Q0)出力は，クロックが2回立ち上がるたびに反転します(**図8-17**)．今回は，このような動作を実現するために2ビットのジョンソン・カウンタを使います．このジョンソン・カウンタのビット0を

グレイ・コード
デコード時にグリッチが出ないことから，ロータリ・エンコーダ用のパターンとしても使われることがある．

グリッチ
バイナリ・リプル・カウンタなどの出力をデコードする際に，デコード出力に現れる非常に短い不必要なパルスのこと．カウンタの出力同士の変化のタイミングの差が発生原因である．

〈図8-17〉5ビット・グレイ・コード・カウンタの動作タイミング・チャート

〈表8-7〉グレイ・コード・カウンタの出力の反転条件(Q0を除く)

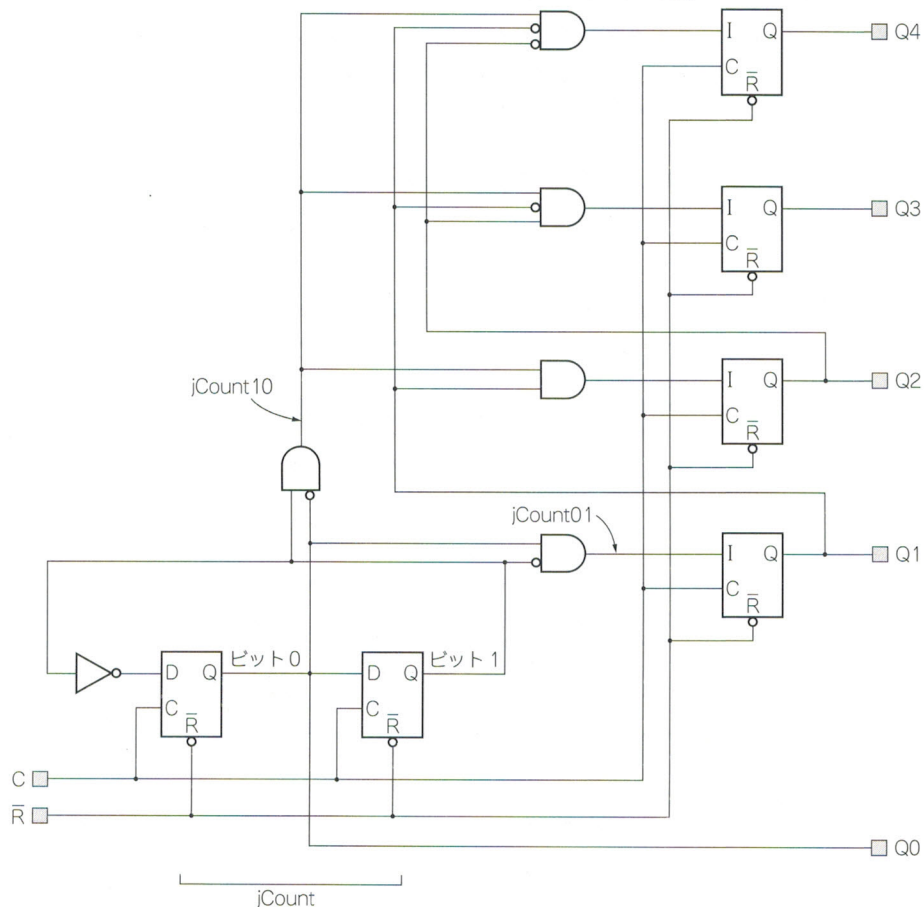
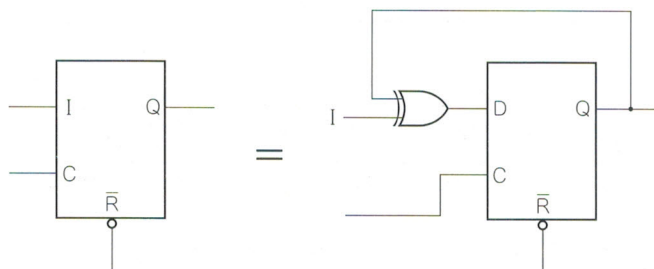
着目する出力ビット	反転条件
MSB（最上位ビット）	jCount10が1でありかつ，MSBより2ビット下からQ1までの出力がすべて0であること
MSBの1ビット下よりQ2出力までの各ビット	jCount10が1でありかつ，着目したビットの1ビット下の出力が1でありかつ，着目したビットより2ビット下よりQ1までのすべての出力が0であること
Q1出力	jCount01が1であること

Q0としてそのまま使用します．

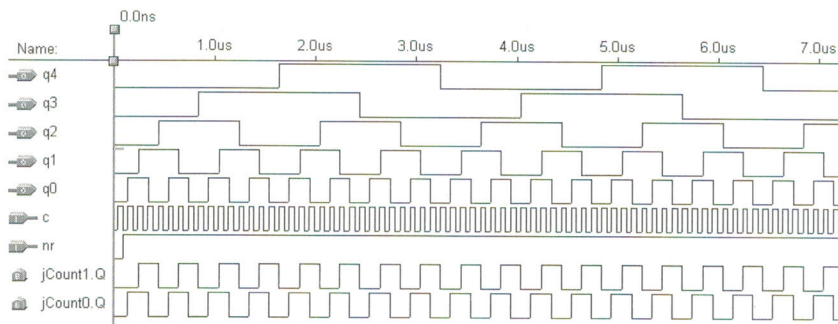
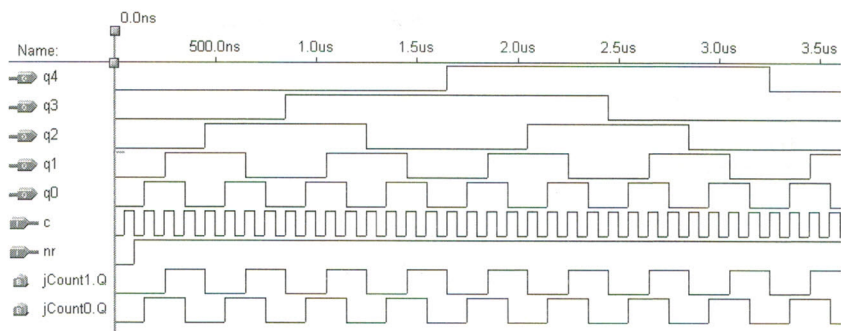
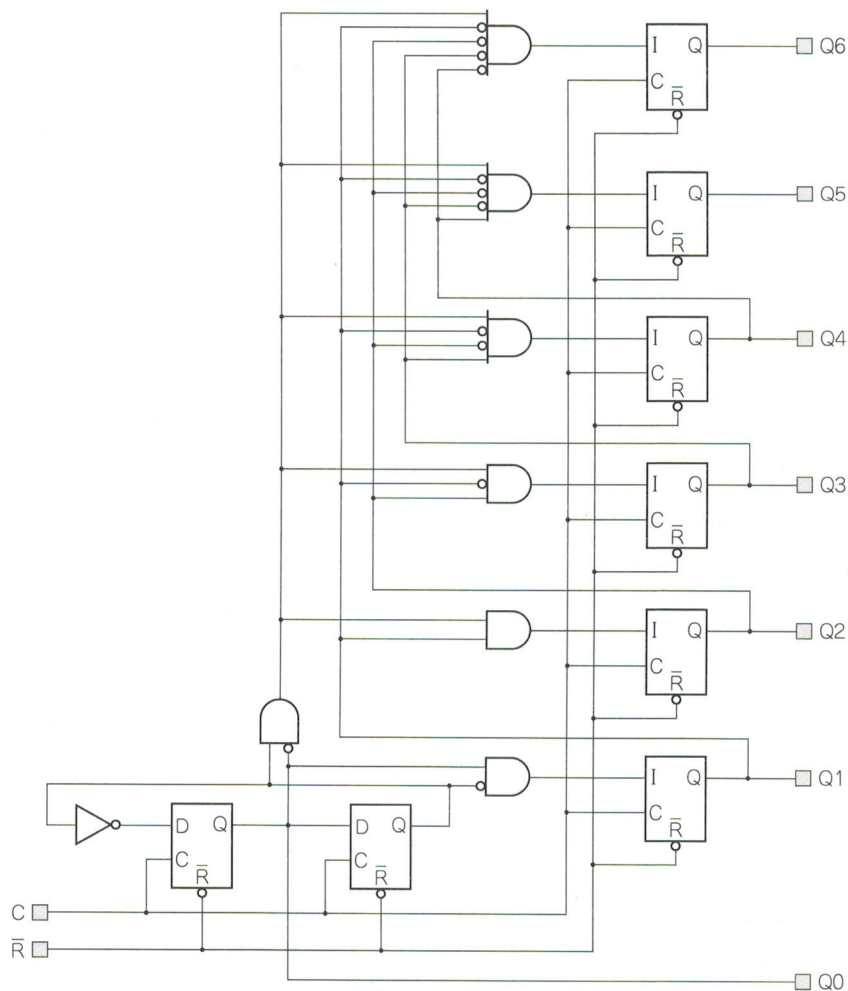
また，設計の都合から，ジョンソン・カウンタの出力をデコードした信号を用意します．jCount01は，ジョンソン・カウンタのビット1が0であり，かつビット0が1のときに1となる信号であり，jCount10はビット1が1でビット0が0のときに1となる信号です．

グレイ・コード・カウンタのQ1以上の出力の反転条件を，**表8-7**に示します．

ここではグレイ・コード・カウンタの各ビットを，**表8-7**のそれぞれの条件

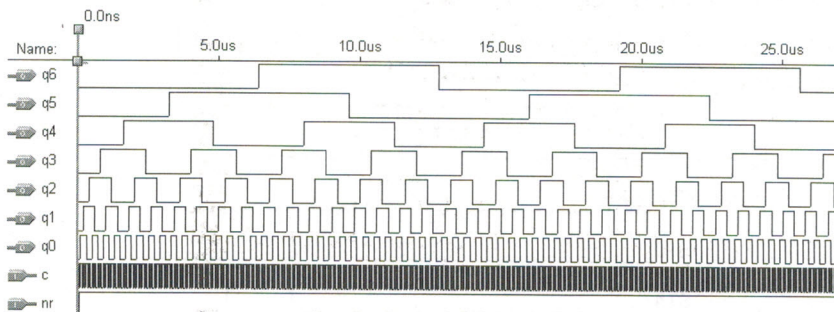
〈図8-18〉5ビット・グレイ・コード・カウンタの回路

〈図8-19〉同期トグル・フリップフロップ(参考)

備考）同期トグル・フリップフロップはI入力が1の時，クロックの立ち上がりで出力が反転する(第5章を参照)

〈リスト8-9〉5ビット・グレイ・コード・カウンタ

```
--
-- 5-bit gray counter
--
library ieee;
use ieee.std_logic_1164.all;

entity grayCount5 is
        port(    c    : in std_logic;
                 nr   : in std_logic;
                 q    : out std_logic_vector(4 downto 0));
end grayCount5;

architecture rtl of grayCount5 is

        signal jCount    : std_logic_vector(1 downto 0);    -- 2-bit jonson counter
        signal jCount01 : std_logic;                         -- jCount = "01"
        signal jCount10 : std_logic;                         -- jCount = "10"

        signal bufQ : std_logic_vector(4 downto 1);         -- gray counter(without bit-0)
        signal invQ : std_logic_vector(4 downto 1);         -- invert pattern

begin

-- ### register discription ###
        syncModule : process(nr,c)
        begin
                if (nr = '0') then
                        jCount <= (others => '0');                  -- asynchronous rest
                        bufQ   <= (others => '0');
                elsif (c'event and c = '1') then
                        jCount <= jCount(0) & (not jCount(1));     -- jonson counter
                        bufQ   <= bufQ xor invQ;                    -- invert gray-counter's bit
                end if;
        end process;

-- ### logic discription ###
-- *** decord jonson counter ***
        jCount10 <= '1' when (jCount = "10") else '0';
        jCount01 <= '1' when (jCount = "01") else '0';

-- *** generate invert pattern ***
        invQ(4) <= jCount10 and (not bufQ(2)) and (not bufQ(1));
        invQ(3) <= jCount10 and bufQ(2) and (not bufQ(1));
        invQ(2) <= jCount10 and bufQ(1);
        invQ(1) <= jCount01;

-- *** assign output data ***
        q(4 downto 1) <= bufQ;
        q(0)          <= jCount(0);

end rtl;
```

〈図8-20〉7ビット・グレイ・コード・カウンタの回路

が満足されたときに，クロックの立ち上がりで反転させています．

フィードバックのさせ方を変えて，フリップフロップを1個節約(この設計方法ではフリップフロップが出力の本数より1個多く必要となる)することも可能なようですが，筆者は設計が容易なこの方法を使っています(**図8-18，リスト8-9，図8-19，図8-20，リスト8-10**)．

〈リスト8-10〉7ビット・グレイ・コード・カウンタ

```
--
-- 7-bit gray counter
--
library ieee;
use ieee.std_logic_1164.all;

entity grayCount7 is
        port(    c    : in std_logic;
                 nr   : in std_logic;
                 q    : out std_logic_vector(6 downto 0));
end grayCount7;

architecture rtl of grayCount7 is

        signal jCount   : std_logic_vector(1 downto 0);   -- 2-bit jonson counter
        signal jCount01 : std_logic;                       -- jCount = "01"
        signal jCount10 : std_logic;                       -- jCount = "10"

        signal bufQ : std_logic_vector(6 downto 1);       -- gray counter(without bit-0)
        signal invQ : std_logic_vector(6 downto 1);       -- invert pattern

begin


-- ### register discription ###
        syncModule : process(nr,c)
        begin
                if (nr = '0') then
                        jCount  <= (others => '0');              -- asynchronous rest
                        bufQ    <= (others => '0');
                elsif (c'event and c = '1') then
                        jCount  <= jCount(0) & (not jCount(1));  -- jonson counter
                        bufQ    <= bufQ xor invQ;                -- invert gray-counter's bit
                end if;
        end process;

-- ### logic discription ###
-- *** decord jonson counter ***
        jCount10 <= '1' when (jCount = "10") else '0';
        jCount01 <= '1' when (jCount = "01") else '0';

-- *** generate invert pattern ***
        invQ(6) <= jCount10 and (not bufQ(4)) and (not bufQ(3)) and (not bufQ(2)) and (not bufQ(1));
        invQ(5) <= jCount10 and bufQ(4) and (not bufQ(3)) and (not bufQ(2)) and (not bufQ(1));
        invQ(4) <= jCount10 and bufQ(3) and (not bufQ(2)) and (not bufQ(1));
        invQ(3) <= jCount10 and bufQ(2) and (not bufQ(1));
        invQ(2) <= jCount10 and bufQ(1);
        invQ(1) <= jCount01;

-- *** assign output data ***
        q(6 downto 1) <= bufQ;
        q(0)          <= jCount(0);

end rtl;
```

ワンホット・シーケンサ

回路の動作の制御
電子回路システムは大きく分けて，データを処理する回路と，それを制御する回路から構成される．

ワンホット・シーケンサ
シーケンサ回路の一種．レジスタ上の1ビットしか'1'にならないように制御されたシフトレジスタとデータの流れを制御する回路から構成される．

ストレート・シーケンサ
1本のまっすぐなシーケンスの流れをもつシーケンサ回路のこと．

シーケンサはディジタル回路において，**回路の動作の制御**に用いられる，いわゆる制御回路の一種です．ここでは，シーケンサの中でも動作が比較的わかりやすい**ワンホット・シーケンサ**を取り上げます．

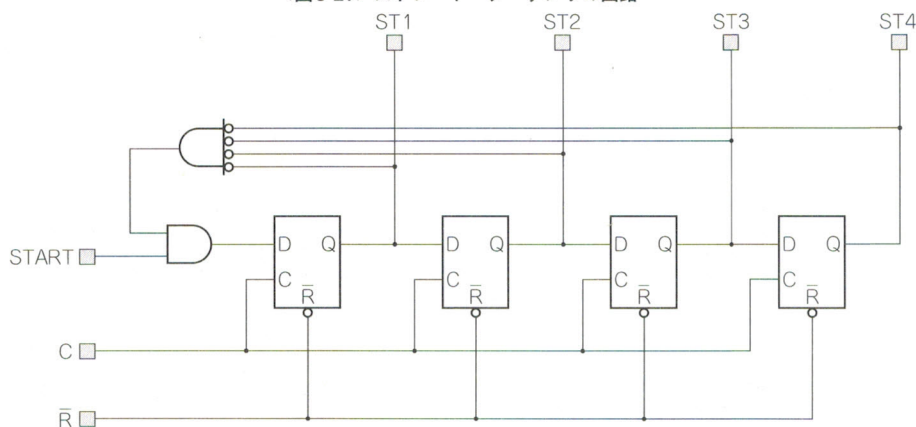
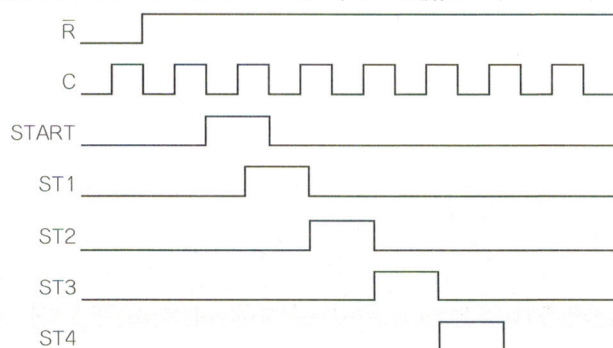
ワンホット・シーケンサは，シフトレジスタをベースとしたシーケンサ回路です．シフトレジスタの場合，複数のステージが1になることがありえますが，ワンホット・シーケンサの場合には，異常が生じでもしない限り1になるのは複数あるステージのうちの1ビットだけです(そうなるように制御する)．ワンホット・シーケンサの由来は，ここから来ているものと思われます．

通常のシフトレジスタと異なるのはこれだけではありません．通常のシフトレジスタではデータの流れは1本だけですし，データが数段前や後方へ飛ぶようなことはありえません．しかしワンホット・シーケンサにおいては，データが走るルートが複数存在したり，データが数段前に戻ってループを形成したり，数段スキップして後方へ飛んだりということが可能です．

これも，シーケンサ全体で1となるのが1ビットであるという条件の下に可能となることです．

①ストレート・シーケンサ

ストレート・シーケンサは，シフトレジスタにレジスタ上の複数ビットが同時

〈図8-21〉ストレート・シーケンサの回路

〈図8-22〉4ステート・ストレート・シーケンサの動作タイミング・チャート

に1とはならないようにする**フィードバック回路**を付加したものです．

4ステートのストレート・シーケンサの回路を**図8-21**に示します．

回路の動作としては，シフトレジスタ内のデータがall0である場合にSTART入力に1が入ると，シフトレジスタの初段のDフリップフロップのD入力が1となります．次のクロックの立ち上がりでST1出力が‘1’となり，シーケンサが起動します．シーケンサの稼働中(ST1～ST4のいずれかが1となっている状態)は，シフトレジスタの初段Dフリップフロップ入力は0となり，シフトレジスタの複数ビットが1となることを防ぎます．

この回路の動作タイミングチャートは**図8-22**のようになります．

とくに何の変哲もない動作で，スタート信号が入ると次のクロックの立ち上がりでステートST1が1となり，これが順に後段へと伝わっていきます．これはまったくシフトレジスタそのものです．

ワンホット・シーケンサにおいては，各ステート出力が1となることでシーケンサがその状態にあることを示します．このストレート・シーケンサの動きを**状態遷移図**で描き表すと，**図8-23**のようになります．

フィードバック回路
帰還回路のこと．

状態遷移図
シーケンサやカウンタなどの回路の状態(出力パターン)がどのように変化していくかを，図にして表したもの．

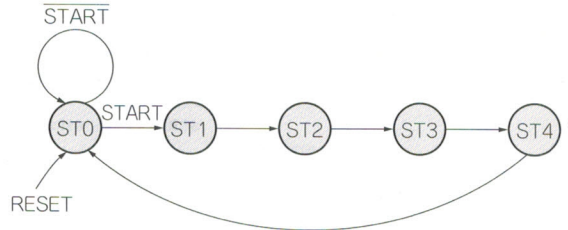
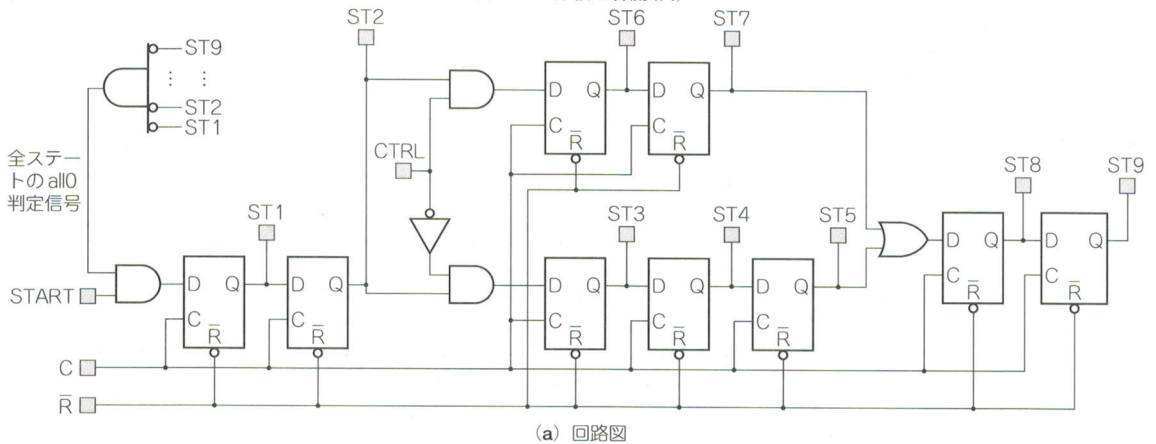
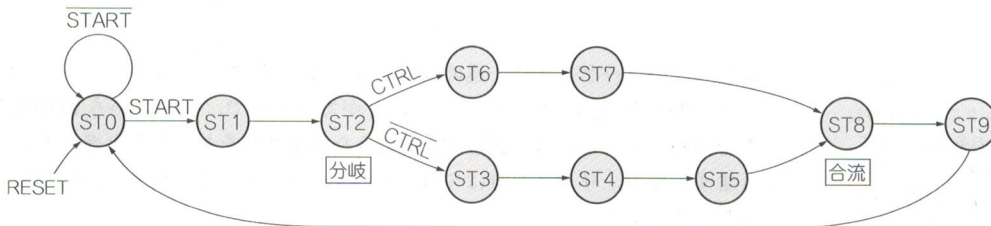
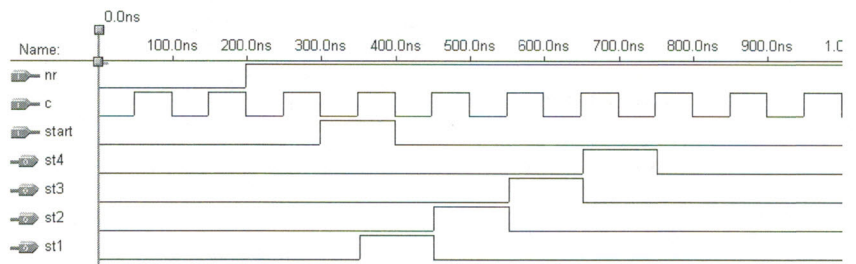
〈図8-23〉4ステート・ストレート・シーケンサの状態遷移図

〈図8-24〉4ステート・ストレート・シーケンサの状態遷移図(2)

〈図8-25〉分枝と合流(例)

(a) 回路図

(b) 状態遷移図

〈リスト8-11〉4ステート・ストレート・シーケンサ

```
--
-- straight sequencer
--
library ieee;
use ieee.std_logic_1164.all;

entity sqA is
        port(   start  : in std_logic;
                c      : in std_logic;
                nr     : in std_logic;
                st1    : out std_logic;
                st2    : out std_logic;
                st3    : out std_logic;
                st4    : out std_logic);
end sqA;

architecture rtl of sqA is

        signal nonState : std_logic;
        signal st1In    : std_logic;
        signal stBuf    : std_logic_vector(4 downto 1);

begin

        process(nr,c)
        begin

                if (nr = '0') then
                        stBuf <= (others => '0');
                elsif (c'event and c = '1') then
                        stBuf(4 downto 1) <= stBuf(3 downto 1) & st1In;
                end if;

        end process;

-- start
        nonState <= '1' when (stBuf = "0000") else '0';
        st1In     <= nonState and start;

        st1 <= stBuf(1);
        st2 <= stBuf(2);
        st3 <= stBuf(3);
        st4 <= stBuf(4);

end rtl;
```

WAITステート
通常，シーケンサ上のステート(状態)は1クロックごとに移動していく．なんらかの条件によって，あるステートに留まるような制御を行うことがあるが，そのような場合このステートをWAITステートと呼ぶ．

また，このシーケンサは一連の動作を終えるとWAIT状態に入り，START信号により動作を再開していると見ることができるので，ST0という**WAITステート**を仮想することにより**図8-24**のような状態遷移をしていると考えることもできます．なお，リセット時はST1～ST4のすべて0となるので，ST0状態に入るものと考えます．

これがもっとも単純な形のワンホット・シーケンサです．こんな簡単な回路で

も，スタート信号をトリガとして一連のパルスを出力する程度の用途には，使うことができます(**リスト8-11**).

分岐と合流
シーケンサのステート(状態)の流れの分岐と合流.

②分枝と合流

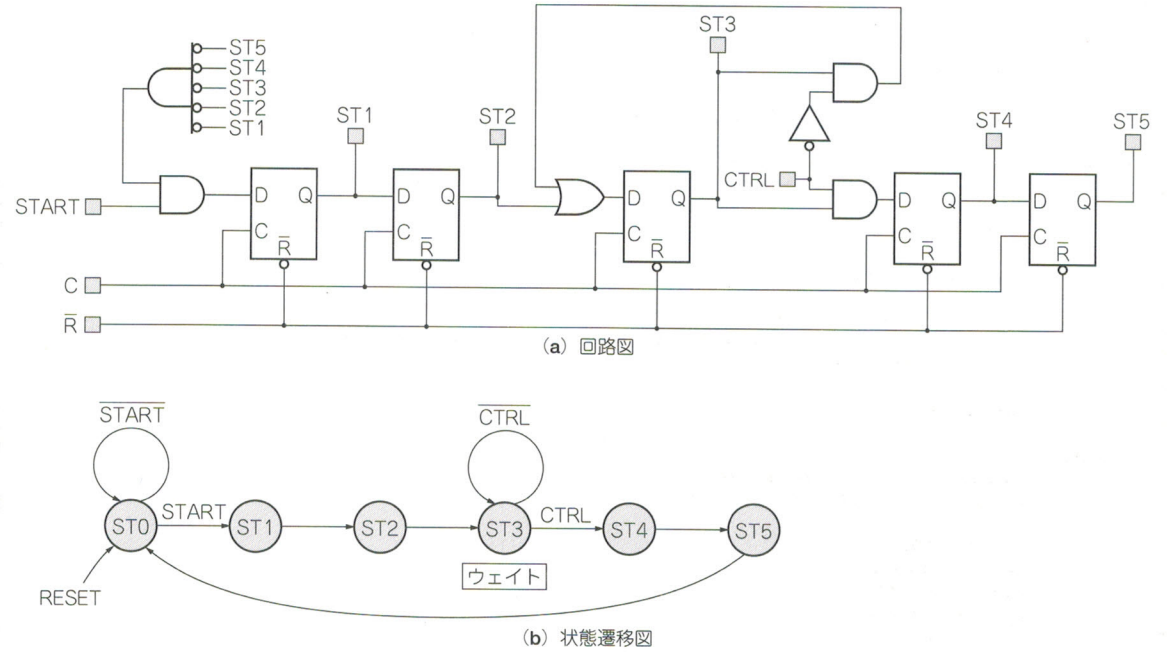
ワンホット・シーケンサ上で，条件により処理を切り換えたいという場合，シーケンスの流れの分枝と合流を使って実現する方法があります．ワンホット・シーケンサにおける流れの分枝には**デマルチプレクサ**を，そして合流にはORゲートを使います(**図8-25**).

デマルチプレクサ
制御信号により一つの入力信号を複数の出力へ振り分ける回路のこと.

ここでは，CTRL入力が0の時にST1→ST2→ST3→ST4→ST5→ST8→ST9という流れで，また，CTRL入力が1の時にはST1→ST2→ST6→ST7→

〈図8-26〉ウェイト・ステートを含む場合

(a) 回路図

(b) 状態遷移図

〈リスト8-12〉分岐と合流を含むシーケンサ

```
--
-- sequencer include branch & join
--
library ieee;
use ieee.std_logic_1164.all;

entity sqB is
        port(   start  : in std_logic;
                ctrl   : in std_logic;
                c      : in std_logic;
                nr     : in std_logic;
                st1    : out std_logic;
                st2    : out std_logic;
                st3    : out std_logic;
                st4    : out std_logic;
                st5    : out std_logic;
                st6    : out std_logic;
                st7    : out std_logic;
                st8    : out std_logic;
                st9    : out std_logic);
```

〈リスト8-12〉分岐と合流を含むシーケンサ(つづき)

```
end sqB;

architecture rtl of sqB is

        signal nonState : std_logic;

        signal st1In : std_logic;
        signal st3In : std_logic;
        signal st6In : std_logic;
        signal st8In : std_logic;

        signal stBuf : std_logic_vector(9 downto 1);

begin

        process(nr,c)
        begin

                if (nr = '0') then
                        stBuf <= (others => '0');
                elsif (c'event and c = '1') then
                        stBuf(2 downto 1) <= stBuf(1) & st1In;
                        stBuf(5 downto 3) <= stBuf(4 downto 3) & st3In;
                        stBuf(7 downto 6) <= stBuf(6) & st6In;
                        stBuf(9 downto 8) <= stBuf(8) & st8In;
                end if;

        end process;

-- start
        nonState <= '1' when (stBuf = "000000000") else '0';
        st1In    <= nonState and start;
-- branch
        st3In <= stBuf(2) and (not ctrl);
        st6In <= stBuf(2) and ctrl;
-- join
        st8In <= stBuf(5) or stBuf(7);

        st1 <= stBuf(1);
        st2 <= stBuf(2);
        st3 <= stBuf(3);
        st4 <= stBuf(4);
        st5 <= stBuf(5);
        st6 <= stBuf(6);
        st7 <= stBuf(7);
        st8 <= stBuf(8);
        st9 <= stBuf(9);

end rtl;
```

ST8→ST9という流れで状態を遷移するシーケンサを取り上げてみました(**リスト8-12**).

③ウェイト・ステート

シーケンサ上で，シーケンスの進行を一時的に停めたいことがあります．このような場合にも，分枝と合流の組み合わせにより対処することが可能です．ウェイト機能をもたせたいステートの出力に，条件ブランチ(デマルチプレクサ)の回路を設け，ウェイトを離脱する条件が満足された場合には，次段をドライブし，

ウェイト・ステート

通常，シーケンサ上のステート(状態)は1クロックごとに移動していく．なんらかの条件によって，あるステートに留まるような制御を行うことがあるが，そのような場合このステートをWAITステートと呼ぶ．

〈図8-27〉ウェイト・ステートのタイマ制御

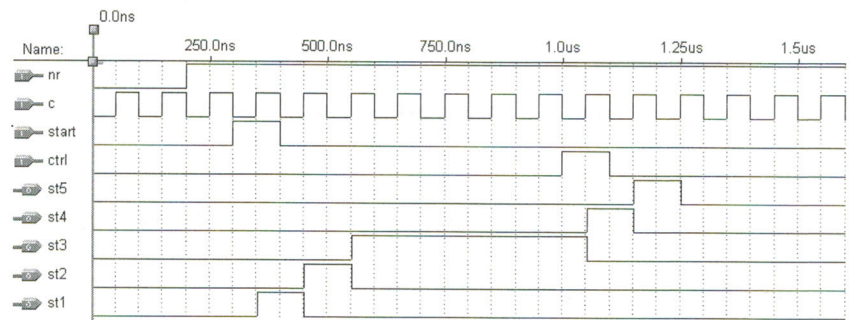
(a) 回路図

(b) 状態遷移図

〈リスト8-13〉ウェイト・ステートを含むシーケンサ

```
--
-- sequencer include wait state
--
library ieee;
use ieee.std_logic_1164.all;

entity sqWaitA is
          port(    start  : in std_logic;
                   ctrl   : in std_logic;
                   c      : in std_logic;
                   nr     : in std_logic;
                   st1    : out std_logic;
                   st2    : out std_logic;
                   st3    : out std_logic;
                   st4    : out std_logic;
                   st5    : out std_logic);
```

〈リスト8-13〉ウェイト・ステートを含むシーケンサ(つづき)

```
end sqWaitA;

architecture rtl of sqWaitA is

          signal nonState  : std_logic;
          signal st1In     : std_logic;
          signal st3In     : std_logic;
          signal st4In     : std_logic;
          signal stBuf     : std_logic_vector(5 downto 1);

begin

          process(nr,c)
          begin

                    if (nr = '0') then
                              stBuf <= (others => '0');
                    elsif (c'event and c = '1') then
                              stBuf(2 downto 1) <= stBuf(1) & st1In;
                              stBuf(3)          <= st3In;
                              stBuf(5 downto 4) <= stBuf(4) & st4In;
                    end if;

          end process;

-- start
          nonState <= '1' when (stBuf = "00000") else '0';
          st1In     <= nonState and start;
-- wait loop
          st3In <= stBuf(2) or (stBuf(3) and (not ctrl) );
-- loop out
          st4In <= stBuf(3) and ctrl;

          st1 <= stBuf(1);
          st2 <= stBuf(2);
          st3 <= stBuf(3);
          st4 <= stBuf(4);
          st5 <= stBuf(5);

end rtl;
```

そうでない場合には自分自身をドライブします(**図8-26**).

自分自身のドライブ信号と前段よりのドライブ信号をOR回路により合流させれば，ウェイト・ステートの実現が可能になります(**リスト8-13**).

④ウェイト・ステートのタイマ制御

ウェイト・ステートの制御は，外部からの制御信号によって行うだけでなく，シーケンサに付属させた**タイマ**(カウンタ)により行うことも可能です．つまり，あらかじめ設定したサイクル数だけ特定のステートを保持し，その後，そのステートを抜け出して以降のステートを実行するわけです(**図8-27**).

タイマ
一定の時間をカウントする回路のこと．ここでは，ダウンカウンタを用いて，クロックが所要の回数入る期間を計っている．

ここではバイナリ・カウンタをタイマ代わりに使います．このカウンタは出力より同期イネーブルへのフィードバック回路により，出力が0以外の値の場合にダウン・カウントを行います．また，**システム・リセット**($\overline{R}$)によりカウンタの値は0に初期化されるため，初期状態ではカウンタは0出力で停止しています．

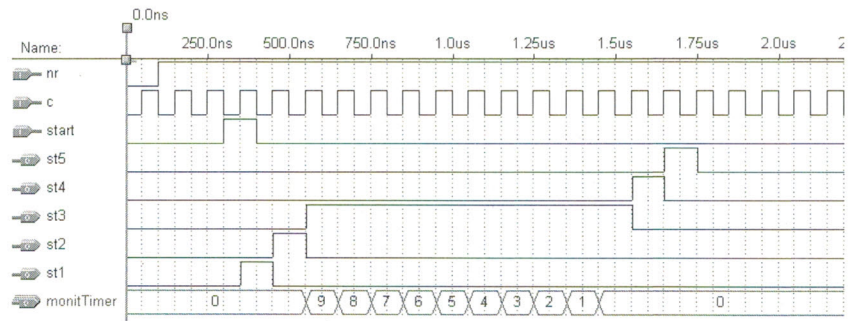
システム・リセット
システムの回路全体を初期状態とするリセット信号のこと．

ウェイト機能をもつステートST3の前段(ステートST2)でダウン・カウンタに9をプリセット(実際には同期プリセットなので，ステートST3への移行と同

〈リスト8-14〉ウェイト・ステートをタイマ制御したシーケンサ

```
--
-- decrement library
--
library ieee;
use ieee.std_logic_1164.all;

package libDec is
          function decrement4(sorce : std_logic_vector(3 downto 0)) return std_logic_vector;
end libDec;

package body libDec is
          function decrement4(sorce : std_logic_vector(3 downto 0)) return std_logic_vector is
                    variable orAll  : std_logic;
                    variable temp   : std_logic_vector(sorce'high downto sorce'low);
          begin
                    orAll := '0';
                    for j in sorce'low to sorce'high loop
                              temp(j)  := sorce(j) xor (not orAll);
                              orAll    := orAll or sorce(j);
                    end loop;
                    return(temp);
          end decrement4;
end libDec;

--
-- sequencer include timer controled wait state
--
library ieee;
use ieee.std_logic_1164.all;
use work.libDec.all;

entity sqWaitB is
          port(    start  : in std_logic;
                   c      : in std_logic;
                   nr     : in std_logic;
                   st1    : out std_logic;
                   st2    : out std_logic;
                   st3    : out std_logic;
                   st4    : out std_logic;
                   st5    : out std_logic;
                    monitTimer : out std_logic_vector(3 downto 0) );
end sqWaitB;

architecture rtl of sqWaitB is

          signal nonState   : std_logic;
          signal st1In      : std_logic;
          signal st3In      : std_logic;
          signal st4In      : std_logic;
          signal stBuf      : std_logic_vector(5 downto 1);

          signal timer      : std_logic_vector(3 downto 0);
          signal timerZero : std_logic;

begin

          process(nr,c)
          begin
```

〈リスト8-14〉ウェイト・ステートをタイマ制御したシーケンサ(つづき)

```
				if (nr = '0') then
						stBuf <= (others => '0');
						timer <= (others => '0');
				elsif (c'event and c = '1') then
						stBuf(2 downto 1) <= stBuf(1) & st1In;
						stBuf(3)             <= st3In;
						stBuf(5 downto 4) <= stBuf(4) & st4In;

						if (stBuf(2) = '1') then
								timer <= "1001";
						elsif (timerZero = '0') then
								timer <= decrement4(timer);
						end if;
				end if;

		end process;

-- start
		nonState <= '1' when (stBuf = "00000") else '0';
		st1In     <= nonState and start;
-- wait loop
		st3In <= stBuf(2) or (stBuf(3) and (not timerZero) );
-- loop out
		st4In <= stBuf(3) and timerZero;

		timerZero <= '1' when (timer = "0000") else '0';

		st1 <= stBuf(1);
		st2 <= stBuf(2);
		st3 <= stBuf(3);
		st4 <= stBuf(4);
		st5 <= stBuf(5);

		monitTimer <= timer;

end rtl;
```

時にプリセットされる).すると,カウンタが起動し,0になるまでダウン・カウントを続けます.

ステートST3は,このダウン・カウンタが0となるまでウェイト状態を保つので,システム・クロック(C)の10サイクル分の期間状態を保持し,その後ステートST4へと移行します.

以上の回路の動作は,すべて**同期制御**なので,制御信号の効果はすべての次のクロックの立ち上がりの時点で現れます(**リスト8-14**).

同期制御
本書は,同期回路設計を前提としている.

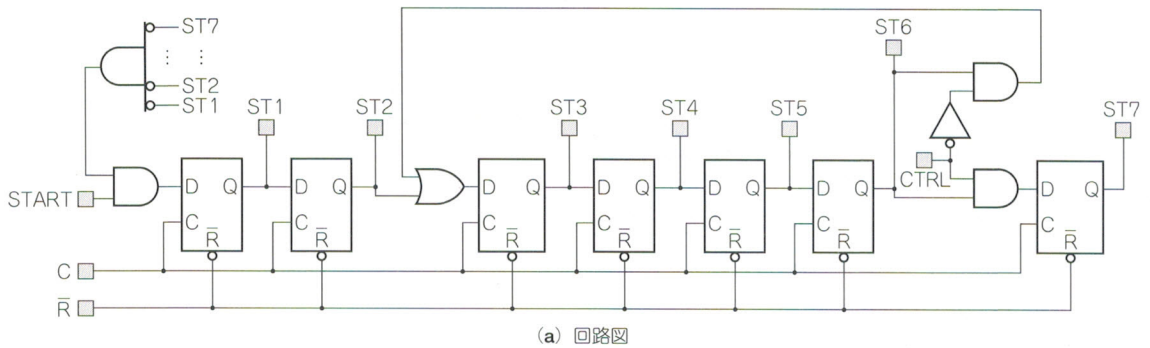
⑤ループ制御

一連の処理を繰り返して実行したいという場合,シーケンサ内部で**ループ制御**を行うことにより,要求に応えることが可能になります.ループ制御とはいって

ループ制御
シーケンス制御を行うにあたって,一連のシーケンスを繰り返し実行すること.

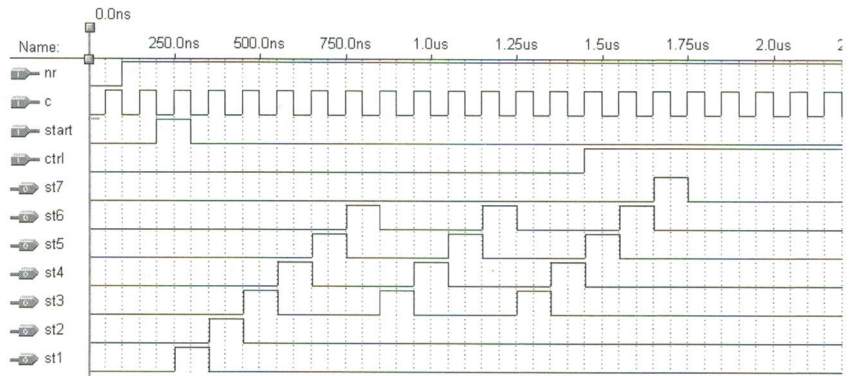
〈図8-28〉ループ制御

(a) 回路図

(b) 状態遷移図

〈リスト8-15〉繰り返しループを含むシーケンサ

```
--
-- sequencer include loop control
--
library ieee;
use ieee.std_logic_1164.all;

entity sqLoopA is
          port(    start  : in std_logic;
                   ctrl   : in std_logic;
                   c      : in std_logic;
                   nr     : in std_logic;
                   st1    : out std_logic;
                   st2    : out std_logic;
                   st3    : out std_logic;
                   st4    : out std_logic;
                   st5    : out std_logic;
                   st6    : out std_logic;
                   st7    : out std_logic);
end sqLoopA;

architecture rtl of sqLoopA is

          signal nonState  : std_logic;
          signal st1In     : std_logic;
          signal st3In     : std_logic;
          signal st7In     : std_logic;
          signal stBuf     : std_logic_vector(7 downto 1);

begin

          process(nr,c)
          begin

                    if (nr = '0') then
                              stBuf <= (others => '0');
                    elsif (c'event and c = '1') then
                              stBuf(2 downto 1) <= stBuf(1) & st1In;
                              stBuf(6 downto 3) <= stBuf(5 downto 3) & st3In;
                              stBuf(7)              <= st7In;
```

〈リスト8-15〉繰り返しループを含むシーケンサ(つづき)

```
                    end if;

          end process;

-- start
          nonState <= '1' when (stBuf = "0000000") else '0';
          st1In     <= nonState and start;
-- wait loop
          st3In <= stBuf(2) or (stBuf(6) and (not ctrl) );
-- loop out
          st7In <= stBuf(6) and ctrl;

          st1 <= stBuf(1);
          st2 <= stBuf(2);
          st3 <= stBuf(3);
          st4 <= stBuf(4);
          st5 <= stBuf(5);
          st6 <= stBuf(6);
          st7 <= stBuf(7);

end rtl;
```

も取り立てて難しいことはなく，先に解説したウェイト・ステートと同様の制御を行えばよいわけです(**図8-28**).

一つだけ異なるのはウェイト・ステートでは，条件により次のステートへと移行しないときには，自分で自分のステートをドライブしていたのに対し，何ステートか前のステートをドライブするようになる点です(**リスト8-15**).

⑥ループ・カウンタによるループ制御

ループ・カウンタ
ループ制御を行う場合に，現在何回目のループを廻っているかを数えるためのカウンタ.

ループ制御に関しても，外部よりの制御に依存するのではなく，**ループ・カウンタ**を設けてループを回す回数を規定することができます．ここでは，前出の回路に3ビットのダウン・カウンタを付加した例を取り上げます.

ウェイト・ステートの制御を行ったときは，カウンタを自走させましたが，ループ制御の場合にはループのなかに複数のステートが存在するため，その中の特定のステートでカウンタを駆動するようにします.

ここでは，ループの直前のステート2でカウンタを4にプリセットし，ループの中のステート4でカウンタを駆動しています(カウント・ダウンさせる)．ループの最後のステートST6の時点でカウンタの値が0の場合には，ループを脱出してステートST7へ進み，カウンタが0でない場合にはステートST3へ戻り，ループ内のシーケンスを繰り返します(**図8-29，リスト8-16**).

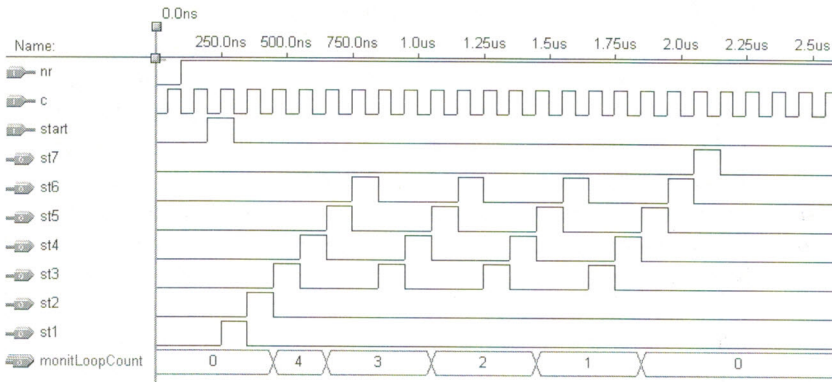
〈図8-29〉ループ・カウンタを使ったループ制御

(a) 回路図

(b) 状態遷移図

〈リスト8-16〉繰り返しループをカウンタで制御したシーケンサ

```
--
-- decrement library
--
library ieee;
use ieee.std_logic_1164.all;

package libDec is
          function decrement3(sorce : std_logic_vector(2 downto 0)) return std_logic_vector;
end libDec;

package body libDec is
          function decrement3(sorce : std_logic_vector(2 downto 0)) return std_logic_vector is
                    variable orAll  : std_logic;
                    variable temp   : std_logic_vector(sorce'high downto sorce'low);
          begin
                    orAll := '0';
                    for j in sorce'low to sorce'high loop
                              temp(j) := sorce(j) xor (not orAll);
                              orAll   := orAll or sorce(j);
                    end loop;
                    return(temp);
          end decrement3;
end libDec;

--
-- sequencer include loop control by loop counter
--
library ieee;
use ieee.std_logic_1164.all;
use work.libDec.all;

entity sqLoopB is
```

```
        port(   start  : in std_logic;
                c      : in std_logic;
                nr     : in std_logic;
                st1    : out std_logic;
                st2    : out std_logic;
                st3    : out std_logic;
                st4    : out std_logic;
                st5    : out std_logic;
                st6    : out std_logic;
                st7    : out std_logic;
                monitLoopCount : out std_logic_vector(2 downto 0) );
end sqLoopB;

architecture rtl of sqLoopB is

        signal nonState  : std_logic;
        signal st1In     : std_logic;
        signal st3In     : std_logic;
        signal st7In     : std_logic;
        signal stBuf     : std_logic_vector(7 downto 1);

        signal loopCount : std_logic_vector(2 downto 0);
        signal loopCountZero : std_logic;

begin

        process(nr,c)
        begin

                if (nr = '0') then
                        stBuf     <= (others => '0');
                        loopCount <= (others => '0');
                elsif (c'event and c = '1') then
                        stBuf(2 downto 1)  <= stBuf(1) & st1In;
                        stBuf(6 downto 3)  <= stBuf(5 downto 3) & st3In;
                        stBuf(7)           <= st7In;

                        if (stBuf(2) = '1') then
                                loopCount <= "100";
                        elsif (stBuf(4) = '1') then
                                loopCount <= decrement3(loopCount);
                        end if;
                end if;

        end process;

-- start
        nonState <= '1' when (stBuf = "0000000") else '0';
        st1In    <= nonState and start;
-- wait loop
        st3In <= stBuf(2) or (stBuf(6) and (not loopCountZero) );
-- loop out
        st7In <= stBuf(6) and loopCountZero;

        loopCountZero <= '1' when (loopCount = "000") else '0';

        st1 <= stBuf(1);
        st2 <= stBuf(2);
        st3 <= stBuf(3);
        st4 <= stBuf(4);
        st5 <= stBuf(5);
        st6 <= stBuf(6);
        st7 <= stBuf(7);

        monitLoopCount <= loopCount;

end rtl;
```

以上のように，ワンホット・シーケンサは，シフトレジスタの基本動作とデータ分枝/合流に関して理解することができれば，設計することができます．まあ，設計と言うよりはフリップフロップにより構成されたレールの上を，1ビットのデータという列車をいかにうまく走らせるかを競う，ゲームかパズルのようなものといえるかもしれません．

もっとも，シーケンサはシステムの動作を決定する重要なパートであるので，くれぐれも列車(1ビットのデータ)の脱線には気をつけましょう(慎重に設計しましょう)．

ウェイト制御，ループ制御信号(CTRL入力)やスタート信号の変化がDフリップフロップのセットアップ時間，ホールド時間の規定に引っ掛かると，当然**メタステーブルが発生**する可能性があります．

メタステーブルが発生
フリップフロップなどの回路において，規定されたセットアップ時間，ホールド時間を守らないと，出力がきちんと変化できずに，異常な状態に陥ることがある(第5章参照)．

■ MAX＋plus IIでVHDLコードをコンパイルする際の注意 ■

MAX + plus II上でVHDLコードをコンパイルしようとする場合には，VHDLコードのファイル名が，VHDLで記述されている機能モジュールのモジュール名と一致していなければなりません．たとえば，circuit1という機能モジュールを記述したファイルは，circuit1.vhd(ただし，".vhd"はVHDLコードを示す拡張子)というファイル名をもたなければなりません．

もし，一致していない場合にはコンパイル時に，その旨のメッセージが出て，エラーとなります．

また，コンパイルにともない，多くのファイルが生成されるので，各デザインごとにフォルダを設けることをお薦めします．

編集雑記

編集部から

● 「賢者は歴史から学び，愚者は経験からも学ばない」とはワイマール憲法時代のビスマルク(ドイツ)の言葉だ．最近，次から次に日本周辺で起こることがらに，まったくあてはまる．

● 狂牛病にはじまって，中国での亡命者の連行，政治とカネ，無認可添加物混入のおびただしい数の食品類，防衛庁の個人情報リスト問題，輸入野菜の残留農薬問題，銀行のシステム障害など….

● 現実を直視しない風土では危機感は生まれないし，危機対応などできない．90年代のバブル崩壊の原因を顧みず，一時の成功体験へ埋没し，失敗に目を背ける傾向は企業社会においても顕著である．

● 成功例の蓄積はしていても，失敗例の蓄積がない．実は失敗例のほうにこそ学ぶべきことが多いものである．なぜ失敗したのか，原因を究明し改善することのほうが重要なのである．ともすると，現実から目を背け，悪いことはすぐ忘れようとする，そして歴史から学ぼうとしない．

● JKフリップフロップの名前の由来がネット上の掲示板にありました．「JはJack，KはKing，QはQueenでJackとKingがQueenを取り合う」という「JackもKingも手を挙げないとQueenはそのまま．JackかKingが手を挙げるとQueenはそちらになびく．両方から御声がかかると，公平を期すために反対側に遷移する」．JackもKingもQueenもいろいろな意味があるので，各自で解釈を….

● 現在進行中の2002 FIFA WORLD CUPはK(Korea)-J(Japan)だけれど，アルファベット順からいってもJ-Kだよな，と思うのは私だけだろか．先々2国共催の場合，政治的にどっちが先だという議論になることは目に見えている．悪しき前例を作ることはやめたほうがよい．

● 今回の特集は，1冊で完結のはずでしたが，予想外にボリュームが膨らみ，次号と共催という形になりました．なお，リストで掲載したファイルはホームページ上で公開する予定です．（檀）

● 米国中間選挙の資金集めとして売り出したブッシュ大統領のブロマイドの売り上げが，140万ドル(約1億7300万円)に達した．写真は3枚組みで，うち1枚は911テロ発生直後，大統領専用機内で電話をかける様子を写したもの…民主党側から「大統領はテロを政治に利用している」とクレームが付いたいわく付きのもの．$150の政治献金と引き換えで支持者に配布されたとか…？米国大統領は“アイドル”並みの人気が実証された．（MASA）

● トランジスタ技術SPECIALの既刊号で紹介しました基板等の頒布サービスを，申し込み締め切り日を過ぎて受け付けているものがありますのでお知らせします．それらは，No.20のMICRO-CAP III，No.23のPALライタ基板，PALASMソフト，No.29のZ80マイコン・キット，No.38の拡張I/Oモジュール・キット，No.55の基板，部品キット，No.57の電源基板キット（第5章を除く）です．申し込み方法は各雑誌掲載のとおりです．

● 本誌掲載記事の利用についてのご注意――本誌掲載記事には著作権があり，また工業所有権が確立されている場合があります．したがって，個人で利用される場合以外は所有者の承諾が必要です．

また，掲載された回路，技術，プログラムを利用して生じたトラブルなどについては，小社ならびに著作権者は責任を負いかねますのでご了承ください．

● ご質問はお手紙で――本誌掲載記事に関する技術的なご質問は，往復はがきか，返信用封筒を同封した書簡を編集部あてお寄せください．執筆者に回送し，直接回答していただきます．質問の内容は当該記事を逸脱しない範囲で，できる限り具体的に明記してください．また，お電話によるご質問にはお応えできませんので，ご了承ください．

次号のお知らせ（9月28日発売）

特集　VHDLによる設計例を紹介
応用・HDL設計学習帳

VHDL記述の基礎を終えたところで，次号ではVHDLによる開発の実際を解説します．CPLDへインプリメントし，ターゲット・ボードを作り，コンフィギュレーション・データを書き込みます．最後にこれらの知識を使って設計例をいくつか紹介します．

トランジスタ技術 エレクトロニクスの基礎と実用技術を満載したフィールド・ワーク・マガジン
SPECIAL No. 79
発行所　CQ出版株式会社（無断転載を禁じます）
〒170-8461　東京都豊島区巣鴨1-14-2
電話　編集部：03(5395)2121　広告部：03(5395)2133
販売部：03(5395)2141
振替　00100-7-10665
編集人　山形孝雄
発行人　蒲生良治　Printed in Japan

2002年7月1日発行　印刷・製本：三晃印刷㈱

ハードウェア・デザイン・シリーズ　好評発売中

身近で便利なワンチップマイコンの使い方と応用
マイクロコントローラAVR入門
加藤　芳夫 著
B5判　160頁
定価1,800円

本書は，AVRというデバイスの紹介を交えながら，特徴・機能・仕様，そしてアーキテクチャを，入門者にもわかりやすく解説しています．

電子機器の設計/製作から調整/評価に役立つ
作りながら学ぶエレクトロニクス測定器
本多　平八郎 著
B5判　280頁
定価1,950円

アーキテクチャ＆命令セットから開発環境，各種応用事例まで
PICマイコン活用ハンドブック
トランジスタ技術編集部 編
B5判　184頁　CD-ROM付き
定価1,850円

フル・カラー　目で見るハードウェア実装の最新技術
エレクトロニクス実装図鑑
トランジスタ技術編集部 編
B5判　208頁
定価1,950円

電力制御のためのデバイスの基礎知識から応用回路まで
パワーMOSFETの実践活用法
トランジスタ技術編集部 編
B5判　180頁
定価1,850円

割り込みとDMAからシリアル/パラレル・ポート，FDD/IDEインターフェースまで
パソコンのレガシィI/O活用大全
桑野　雅彦 著
B5判　152頁　FD付き
定価1,800円

抵抗，コンデンサ，コイル，ダイオード，線材＆コネクタの機能と特徴
受動部品の選び方と活用ノウハウ
トランジスタ技術編集部 編
B5判　144頁
定価1,800円

A-D/D-Aコンバータの接続方法からディジタル・フィルタの実現まで
DSPのハードウェアと信号処理の実際
トランジスタ技術編集部 編
B5判　144頁
定価1,800円

発光ダイオードからフォト・カプラ，赤外線，光ファイバの応用まで
光エレクトロニクスの基礎と活用法
トランジスタ技術編集部 編
B5判　162頁
定価1,680円

シリアル・ポートとパラレル・ポートを活用しよう
パソコン・アダプタの製作＆応用
トランジスタ技術編集部 編
B5判　144頁　FD付き
定価1,800円

部品がわかればハードウェア技術がわかる
わかる電子回路部品 完全図鑑
トランジスタ技術編集部 編
B5判　160頁 フルカラー
定価1,890円

抵抗，コンデンサ，インダクタ，機構部品の特徴と仕様
わかる電子部品の基礎と活用法
薊　利明／竹田　俊夫 著
B5判　184頁
定価1,733円

CQ出版社　〒170-8461 東京都豊島区巣鴨1-14-2 販売部 ☎(03)5395-2141 振替00100-7-10665

（定価は税込）　CQ出版社のホームページURL http://www.cqpub.co.jp/

ISBN4-7898-3740-8

C3055 ¥1752E

CQ出版社

定価：本体1,752円（税別）

トランジスタ技術
SPECIAL